Alpha Programming (Part 1)
CS 347
Feb 19 & 24
1998

Topics

- Basics
- Control Flow
- Procedures
- Instruction Formats

Alpha Processors

Reduced Instruction Set Computer (RISC)

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

Assumes compiler will do optimizations

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

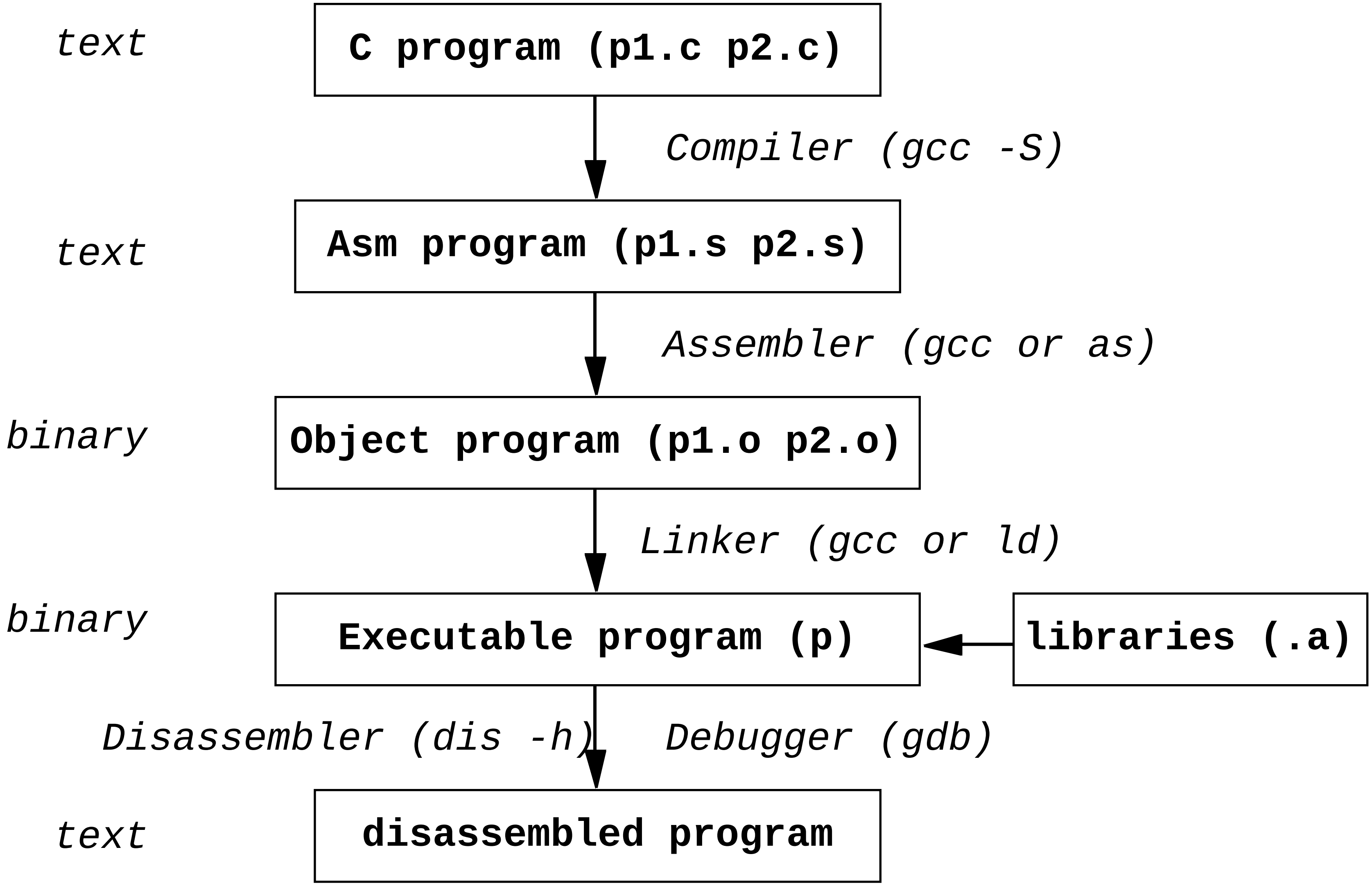
A 2nd Generation RISC Instruction Set Architecture

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

Very High Performance Machines

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

Translation Process



text **C program (p1.c p2.c)**

Compiler (gcc -S)

text **Asm program (p1.s p2.s)**

Assembler (gcc or as)

binary **Object program (p1.o p2.o)**

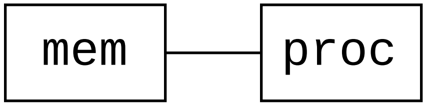
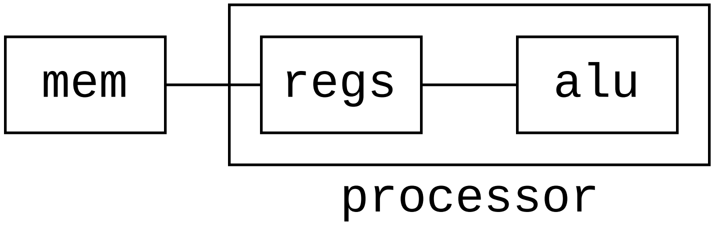
Linker (gcc or ld)

binary **Executable program (p)** ← **libraries (.a)**

Disassembler (dis -h) *Debugger (gdb)*

text **disassembled program**

Abstract Machines

Machine Model	Data	Control
C  	1) char 2) int, float 3) double, long 4) struct, array 5) pointer	1) loops 2) conditionals 3) goto 4) Proc. call 5) Proc. return
ASM  	1) byte 2) 4-byte word 3) 8-byte word 4) contiguous word allocation 5) address of initial byte	3) branch/jump 4) jump & link

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

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

Registers (cont.)

Important Ones for Now

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

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

Program Representations

C Code

```
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

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

Prog. Representation (Cont.)

Object

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

Disassembled

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

Run gdb on object code

```
x/10 0x120001130
```

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

```
dissassemble test1
```

- **Print disassembled version of procedure**

Alternate Disassembly

Alpha program “dis”

`dis file.o`

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

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

Returning a Value from a Procedure

C Code

```
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 $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

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

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

Annotated Assembly

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

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

Array Indexing

C Code

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

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

Annotated Assembly

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

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

Array Indexing (Cont.)

C Code

```
long int garray[10];

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

Annotated Assembly

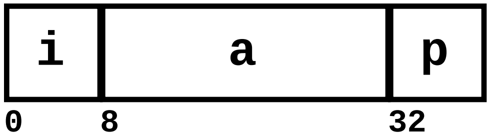
```
         .comm   garray,80

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

Structures & Pointers

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



C Code

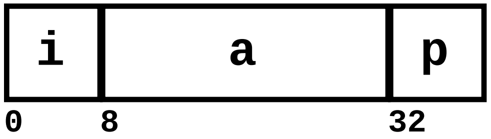
```
void
set_i(struct rec *r,
      long int val)
{
  r->i = val;
}
```

Annotated Assembly

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

Structures & Pointers (Cont.)

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



C Code

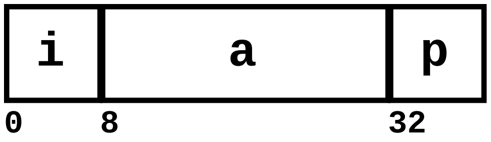
```
long int *
find_a(struct rec *r,
       long int idx)
{
  return &r->a[idx];
}
```

Annotated Assembly

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

Structures & Pointers (Cont.)

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



C Code

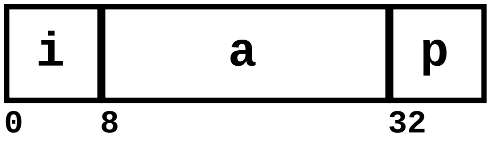
```
void
set_p(struct rec *r,
      long int *ptr)
{
  r->p = ptr;
}
```

Annotated Assembly

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

Structures & Pointers (Cont.)

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



C Code

```
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

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

– *Cond* : branch condition, relative to zero

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

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

Unconditional Branches

`br` ***`label`***

Conditional Branches

Comparison Instructions

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

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

C Code

```
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

```
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:

`cmov`*`Cond`*`Ra, Rb, Rc`

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

Psuedo-code example:

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

Conditional Move Example

C Code

```
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

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

Annotated Assembly

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

“While” Loop Example

C Code

```
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

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

direct translation

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

“For” Loop Example

C Code

```
/* 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;
}
```

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

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

Annotated Assembly

```
amax:
        ldq $0,0($16)    # result = a[0]
        bis $31,1,$3     # i = 1
        cmplt $3,$17,$1  # if (i >= count),
        beq $1,$61       #  branch to 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
```

Jumps

Characteristics:

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

Format:

`jmp Ra,(Rb),`*`Hint`*

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

Compiling Switch Statements

C Code

```
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

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

Assembly: Setup

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

Jump Table

Table Contents

```
$74:
     .gprel32 $68
     .gprel32 $69
     .gprel32 $70
     .gprel32 $71
     .gprel32 $72
     .gprel32 $73
```

Enumerated Values

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

Targets & Completion

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

Procedure Calls & Returns

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

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

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

Annotated Assembly

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

Stack-Based Languages

Languages that support recursion

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

Stack Allocated in *Frames*

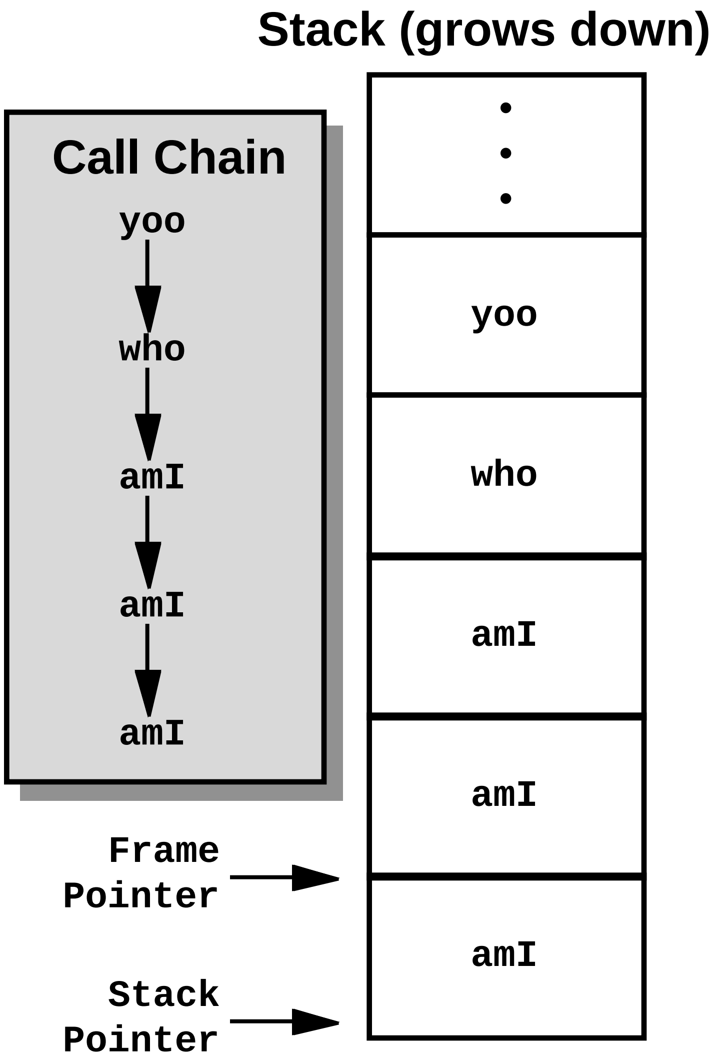
- **state for procedure invocation**
  - return point, arguments, locals

Code Example

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

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

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



Stack (grows down)

Call Chain

`yoo` → `who` → `amI` → `amI` → `amI`

Stack:

- •
•
•
- **yoo**
- **who**
- **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**

```
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

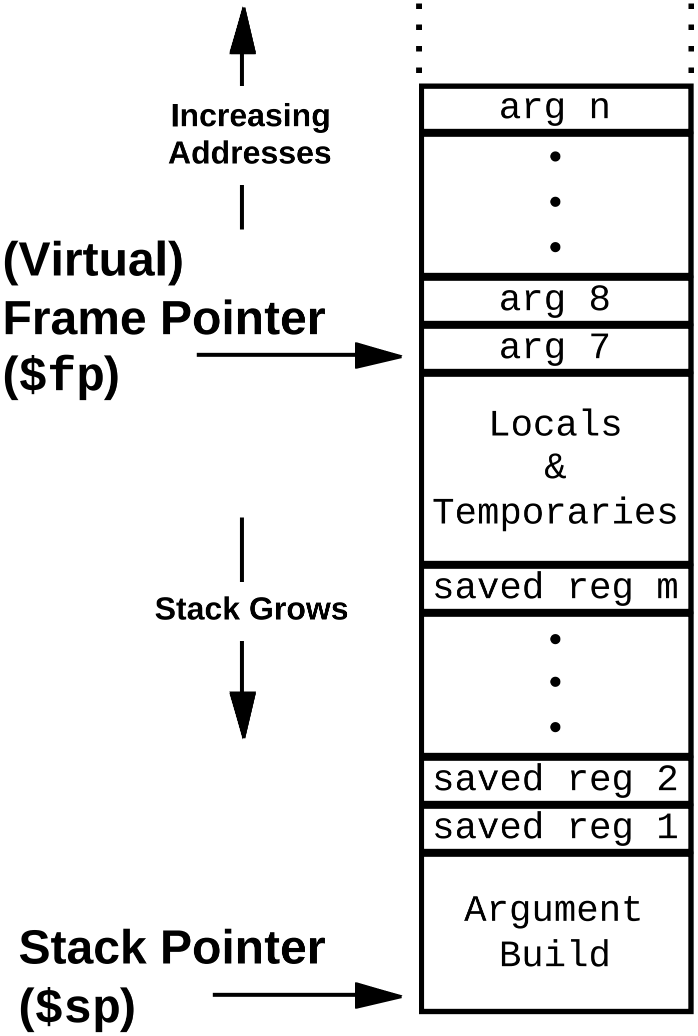
Alpha Stack Frame

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**



Increasing Addresses

(Virtual) Frame Pointer (`$fp`)

Stack Grows

Stack Pointer (`$sp`)

`arg n`

•
•
•

`arg 8`

`arg 7`

`Locals & Temporaries`

`saved reg m`

•
•
•

`saved reg 2`

`saved reg 1`

`Argument Build`

Stack Frame Requirements

Procedure Categories

- **Leaf procedures that do not use stack**
  - Do not call other procedures
  - Can fit all temporaries in caller-save registers
- **Leaf procedures that use stack**
  - Do not call other procedures
  - Need stack for temporaries
- **Non-leaf procedures**
  - Must use stack (at the very least, to save the return address ($26))

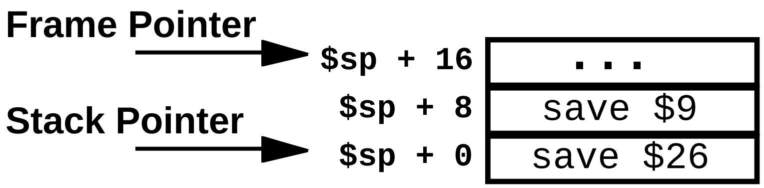
Stack Frame Structure

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

Stack Frame Example

C Code

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



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

Procedure Prologue

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

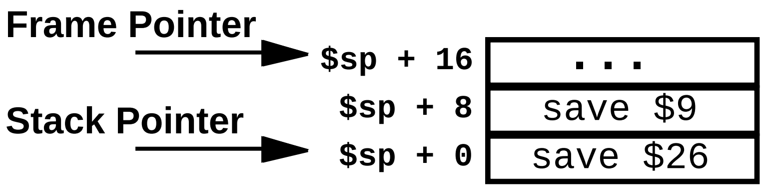
Procedure Epilogue

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

Stack Frame Example (Cont.)

C Code

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



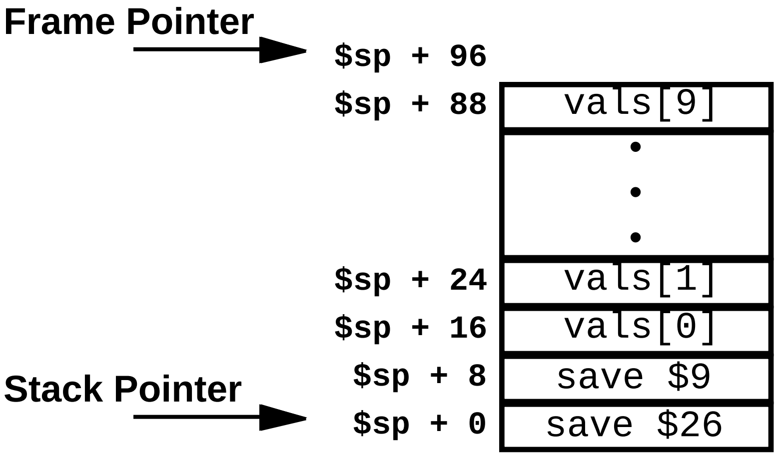
Annotated Assembly

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

Stack Frame Example #2

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]);
}
```



- **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
```

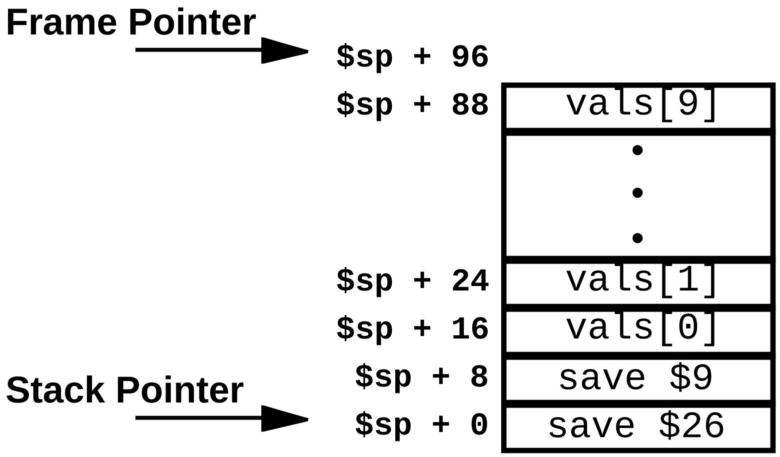
Stack Frame Example #2 (Cont.)

C Code

```
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
```



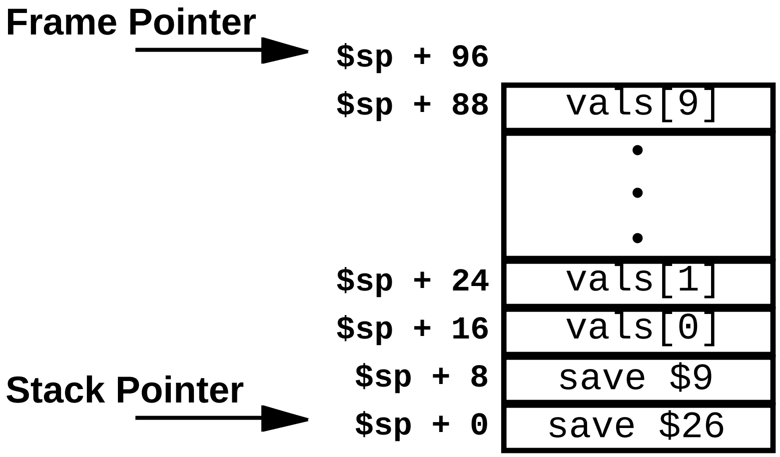
Procedure Epilogue

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

Stack Frame Example #2 (Cont.)

C Code

```
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]);
}
```



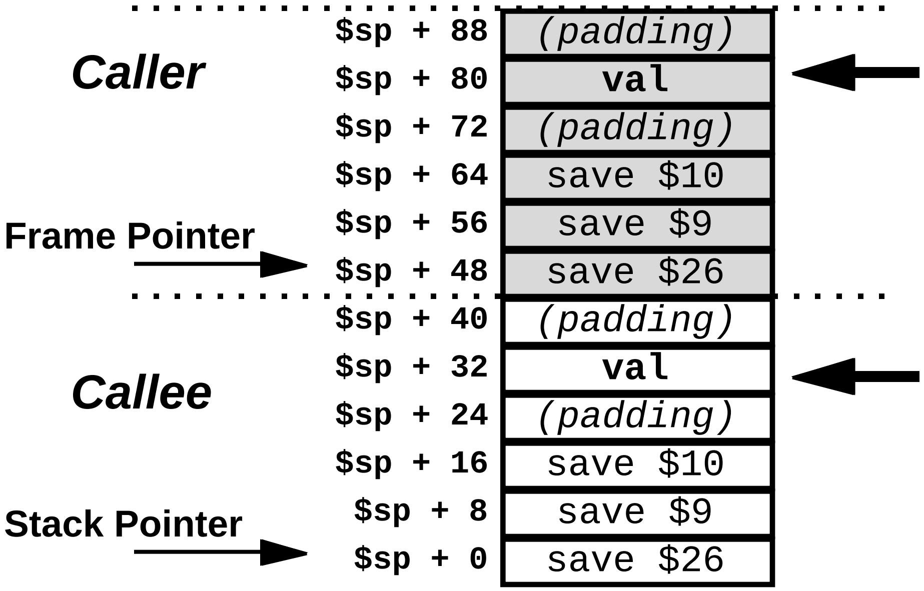
Procedure Body

```
        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
```

Stack Addrs as Procedure Args

C Code

```
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;
}
```



Caller

Frame Pointer

$sp + 88	*(padding)*
$sp + 80	**val**
$sp + 72	*(padding)*
$sp + 64	save $10
$sp + 56	save $9
$sp + 48	save $26
$sp + 40	*(padding)*
$sp + 32	**val**
$sp + 24	*(padding)*
$sp + 16	save $10
$sp + 8	save $9
$sp + 0	save $26

Callee

Stack Pointer

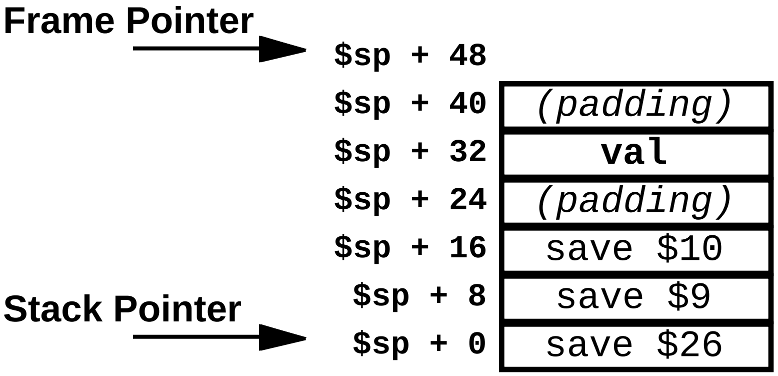
- **Stack frame: 48 bytes**
- **Padded to 16B alignment**
- **`val` stored at $sp + 32**
- **“$sp + 32” passed as second argument ($17) to recursive call of `rfact2`**

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

Stack Addrs as Procedure Args (Cont.)

C Code

```
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;
}
```



Frame Pointer

Stack Pointer

$sp + 48	
$sp + 40	*(padding)*
$sp + 32	**val**
$sp + 24	*(padding)*
$sp + 16	save $10
$sp + 8	save $9
$sp + 0	save $26

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

```
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;
}
```

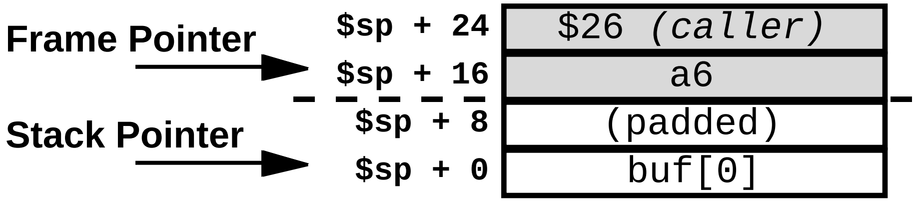
```
void crash()
{
  overwrite(0,0,0,0,0,0,0);
}
```

This code results in a segmentation fault on the Alpha!

Stack Corruption Example (Cont.)

C Code

```
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

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

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

-> overwrites caller stack frame!

Instruction Formats

Arithmetic Operations:

- **all register operands**
  - `addq $1, $7, $5`

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

- **with a literal operand**
  - `addq $1, 15, $5`

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

Branches:

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

6	5	21
Opcode	Ra	Displacement

Jumps:

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

6	5	5	16
Opcode	Ra	Rb	Hint

Loads & Stores:

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

6	5	5	16
Opcode	Ra	Rb	Offset