

# Instruction Set Comparisons

## CS 740

Sept. 15, 1999

### Topics

- **Code Examples**
  - Procedure Linkage
  - Sparse Matrix Code
- **Instruction Sets**
  - Alpha
  - PowerPC
  - VAX

## Procedure Examples

### Procedure linkage

- Passing of control and data
- stack management

### Control

- Register tests
- Condition codes
- loop counters

### Code Example

```
int rfact(int n)
{
    if (n <= 1)
        return 1;
    return n*rfact(n-1);
}
```

# Alpha rfact

## Registers

- \$16 Argument n
- \$9 Saved n
  - Callee save
- \$0 Return Value

## Stack Frame

- 16 bytes
- \$9
- \$26 return PC

\$sp + 8	save \$9
\$sp + 0	save \$26

```
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 = n
           cmple $9,1,$1     # if (n <= 1) then
           bne $1,$80       # branch to $80
           subq $9,1,$16     # $16 = n - 1
           bsr $26,rfact..ng # recursive call
           mulq $9,$0,$0     # $0 = n*rfact(n-1)
           br $31,$81       # branch to epilogue
           .align 4
$80:      bis $31,1,$0       # return val = 1
$81:      ldq $26,0($30)     # restore retnr addr
           ldq $9,8($30)     # restore $9
           addq $30,16,$30   # $sp += 16
           ret $31,($26),1
```

# VAX

## Pinnacle of CISC

- Maximize instruction density
- Provide instructions closely matched to typical program operations

## Instruction format

- OP, arg1, arg2, ...
  - Each argument has arbitrary specifier
  - Accessing operands may have side effects

## Condition Codes

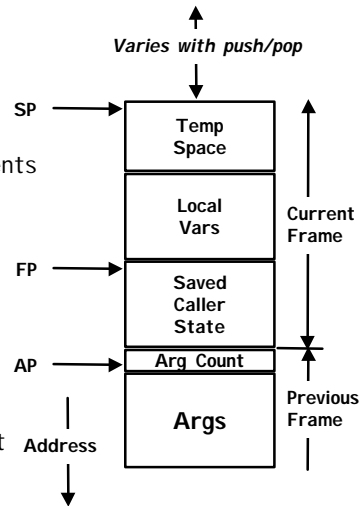
- Set by arithmetic and comparison instructions
- Basis for successive branches

## Procedure Linkage

- Direct implementation of stack discipline

## VAX Registers

- **R0–R11**    **General purpose**
  - R2–R7    Callee save in example code
  - Use pair to hold double
- **R12**    **AP**    **Argument pointer**
  - Stack region holding procedure arguments
- **R13**    **FP**    **Frame pointer**
  - Base of current stack frame
- **R14**    **SP**    **Stack pointer**
  - Top of stack
- **R15**    **PC**    **Program counter**
  - Used to access data in code
- **N C V Z**    **Condition codes**
  - Information about last operation result
  - Negative, Carry, 2's OVF, Zero



– 5 –

CS 740 F 99

## VAX Operand Specifiers

### Forms

Notation	Value	Side Eff	Use
<b>Ri</b>	<b>ri</b>		<b>General purpose register</b>
<b>\$v</b>	<b>v</b>		<b>Immediate data</b>
<b>(Ri)</b>	<b>M[ri]</b>		<b>Memory reference</b>
<b>v(Ri)</b>	<b>M[ri+v]</b>		<b>Mem. ref. with displacement</b>
<b>A[Ri]</b>	<b>M[a+ri*d]</b>		<b>Array indexing</b>
		– A is specifier denoting address a	
		– d is size of datum	
<b>(Ri)+</b>	<b>M[ri]</b>	<b>Ri += d</b>	<b>Stepping pointer forward</b>
<b>-(Ri)</b>	<b>M[ri-d]</b>	<b>Ri -= d</b>	<b>Stepping pointer back</b>

### Examples

- Push *src*                    `move src, -(SP)`
- Pop *dest*                    `move (SP)+, dest`

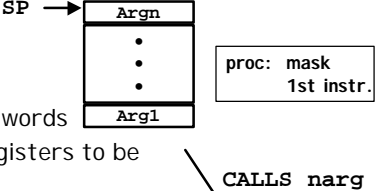
– 6 –

CS 740 F 99

# VAX Procedure Linkage

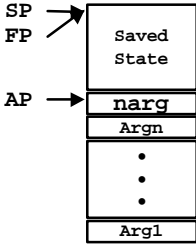
## Caller

- Push Arguments
  - Relative to SP
- Execute CALLS narg, proc
  - narg denotes number of argument words
  - proc starts with mask denoting registers to be saved on stack



## CALLS Instruction

- Creates stack frame
  - Saved registers, PC, FP, AP, mask, PSW
- Sets AP, FP, SP



## Callee

- Compute return value in R0
- Execute ret
  - Undoes effect of CALLS

# VAX rfact

## Registers

- r6 saved n
- r0 return value

## Stack Frame

- save r6

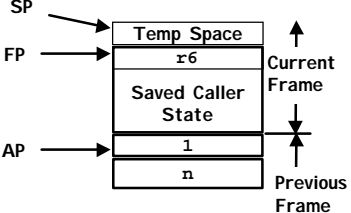
## Note

- Destination argument last

```

_rfact:
    .word 0x40      # Save register r6
    movl 4(ap),r6  # r6 <- n
    cmpl r6,$1    # if n > 1
    jgtr L1       # then goto L1
    movl $1,r0    # r0 <- 1
    ret          # return

L1:
    pushab -1(r6) # push n-1
    calls $1,_rfact # call recursively
    mull2 r6,r0   # return result * n
    ret
    
```



## Sparse Matrix Code

### Task

- **Multiply sparse matrix times dense vector**
  - Matrix has many zero entries
  - Save space and time by keeping only nonzero entries
  - Common application

### Compressed Sparse Row Representation

$\begin{bmatrix} 3.5 & 0.9 & & 2.2 \\ & 4.1 & & 1.9 \\ 4.6 & & 0.72 & 2.7 & 3.0 \\ & & & 2.9 \\ & & 1.2 & & 2.8 \\ & & & & & 3.4 \end{bmatrix}$	$\begin{aligned} & [(0, 3.5) (1, 0.9) (3, 2.2)] \\ & [(1, 4.1) (3, 1.9)] \\ & [(0, 4.6) (2, 0.7) (3, 2.7) (5, 3.0)] \\ & [(2, 2.9)] \\ & [(2, 1.2) (4, 2.8)] \\ & [(5, 3.4)] \end{aligned}$
---	---

## CSR Encoding

### Parameters

- **nrow**      Number of rows (and columns)
- **nentries**    Number of nonzero matrix entries

### Val

- List of nonzero values      (nentries)

### Cindex

- List of column indices      (nentries)

### Rstart

- List of starting positions for each row      (nrow+1)

```
typedef struct {
    int nrow;
    int nentries;
    double *val;
    int *cindex;
    int *rstart;
} csr_rec, *csr_ptr;
```

## CSR Example

### Parameters

- nrow = 6
- nentries = 13

### Val

[3.5, 0.9, 2.2, 4.1, 1.9, 4.6, 0.7, 2.7, 3.0, 2.9, 1.2, 2.8, 3.4]

### Cindex

[0, 1, 3, 1, 3, 0, 2, 3, 5, 2, 2, 4, 5]

### Rstart

[0, 3, 5, 9, 10, 12, 13]

```

[(0,3.5) (1,0.9) (3,2.2)]
[(1,4.1) (3,1.9)]
[(0,4.6) (2,0.7) (3,2.7) (5,3.0)]
[(2,2.9)]
[(2,1.2) (4,2.8)]
[(5,3.4)]
    
```

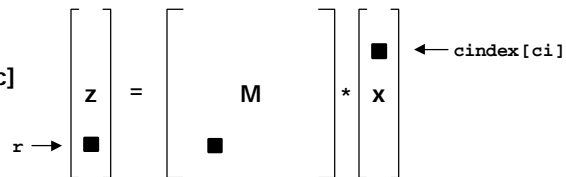
## CSR Multiply: Clean Version

```

void csr_mult_smpl(csr_ptr M, double *x, double *z)
{
    int r, ci;
    for (r = 0; r < M->nrow; r++) {
        z[r] = 0.0;
        for (ci = M->rstart[r]; ci < M->rstart[r+1]; ci++)
            z[r] += M->val[ci] * x[M->cindex[ci]];
    }
}
    
```

### Innermost Operation

- $z[r] += M[r,c] * x[c]$
- Column  $c$  given by  $cindex[ci]$
- Matrix element  $M[r,c]$  by  $val[ci]$



## CSR Multiply: Fast Version

```
void csr_mult_opt(csr_ptr M, ftype_t *x, ftype_t *z)
{
    ftype_t *val = M->val;
    int *cindex_start = M->cindex;
    int *cindex = M->cindex;
    int *rnstart = M->rstart+1;
    ftype_t *z_end = z+M->nrow;

    while (z < z_end) {
        ftype_t temp = 0.0;
        int *cindex_end = cindex_start + *(rnstart++);
        while (cindex < cindex_end)
            temp += *(val++) * x[*cindex++];
        *z++ = temp;
    }
}
```

### Performance

- Approx 2X faster
- Avoids repeated memory references

- 13 -

CS 740 F99

## Optimized Inner Loop

```
while (...)
    temp += *(valp++) * x[*cip++];
```

### Inner Loop Pointers

**cip**            steps through cindex

**valp**          steps through Val

Multiply next matrix value by vector element and add to sum

- 14 -

CS 740 F99

## VAX Inner Loop

### Registers

- r4 cip
- r2,r3 temp
- r5 valp
- r6 cip\_end
- r10 x

```
while (...)  
    temp += *(valp++) * x[*cip++];
```

### Observe

- muld3 instruction does 1/2 of the work!

```
L36:  
    movl (r4)+,r0    # r0 <- *cip++  
    muld3 (r5)+,(r10)[r0],r0 # r0,r1 <- *valp++ * x[r0]  
    addd2 r0,r2      # temp += r0,r1  
    cmpl r4,r6       # if not done  
    jlssu L36        # then goto L36
```

## Power / PowerPC

### History

- **IBM develops Power architecture**
  - Basis of RS6000
  - Somewhere between RISC & CISC
- **IBM / Motorola / Apple combine to develop PowerPC architecture**
  - Derivative of Power
  - Used in Power Macintosh

### CISC-like features

- **Registers with control information**
  - Set of condition registers (CRO-7) holding outcome of comparisons
  - link register (LR) to hold return PC
  - count register (CTR) to hold loop count
- **Updating load / stores**
  - Update base register with effective address

## PowerPC Curiosities

### Loop Counter

```
mtspr CTR r3
  » CTR <-- r3
bc CTR=0, loop
  » CTR--
  » If (CTR == 0) goto loop
```

### Updating Load/Store

```
lu r3, 4(r11)
  » EA <-- r11 + 4
  » r3 <-- M[EA]
  » r11 <-- EA
```

### Multiply/Accumulate

```
fma fp3, fp1, fp0, fp3
  » fp3 <-- fp1 * fp0 + fp3
```

- 17 -

CS 740 F 99

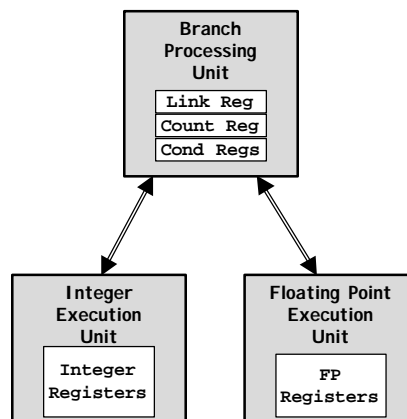
## PowerPC Structure

### System Partitioning

- **Branch Unit**
  - Fetch instructions
  - Make control decisions
- **Integer Unit**
  - Integer & address computations
- **Floating Point Unit**
  - Floating Pt. computations

### Register State

- Partitioned like system
- Allows units to operate autonomously



- 18 -

CS 740 F 99

## IBM Compiler PPC Inner Loop

### Registers

- r3 \*cip
- r4 x
- r10 valp-8
- r11 cip
- fp3 temp
- CNT # iterations

```
while (...)
    temp += *(valp++) * x[*cip++];
```

```
__L208:
    rlinm    r3,r3,3,0,28    # *cip * 8
    lfdx    fp0,r4,r3      # fp0 <- x[*cip]
    lfdx    fp1,8(r10)     # fp1 <- *(++ valp)
    lu      r3,4(r11)      # r3 <- *** cip
    fma     fp3,fp1,fp0,fp3 # fp3 += fp1*fp0
    # Decrement & loop
    bc     BO_dCTR_NZERO,CR0_LT,__L208
```

### Observations

- Makes good use of PPC features
  - Multiply-Add
  - Updating loads
  - Loop counter
- Requires sophisticated compiler
  - Converted p++ into ++p
  - Determine loop count *a priori*

- 19 -

CS 740 F99

## CodeWarrior Compiler PPC Inner Loop

### Registers

- r4 x
- r3 valp
- r8 cip
- fp2 temp

```
while (...)
    temp += *(valp++) * x[*cip++];
```

```
lwz    r0,0(r8)      # r0 = *cip
addi   r8,r8,4      # cip++
lfd    fp1,0(r3)    # fp1 = *valp
addi   r3,r3,8      # valp++
slwi   r0,r0,3      # r0 *= 8
lfdx   fp0,r4,r0    # fp0 = x[]
fmadd  fp2,fp1,fp0,fp2 # temp += () * ()
cmplw  r8,r10      # Compare r8 : r0?
blt    *-32        # Loop if <
```

### Observations

- Limited use of PPC features
  - Multiply-Add
- High performance on modern machines
  - They can do lots of things at once
  - Instruction ordering less critical

- 20 -

CS 740 F99

## Performance Comparison

### Experiment

- 10 X 10 matrices
- 100% density
- 100 multiply accumulates

Machine	MHz	msecs	Cyc/El	Compiler
VAX	25?	2448	122?	GCC
MIPS	25	365	18	GCC
PPC 601	62	63	8	IBM
Pentium	90	79	11	GCC
HP Precision	100	50	10	GCC
UltraSparc	160	38	12.5	GCC
PPC 604e	200	14	5.6	CodeWarrior
MIPS R10000	185	13.4	5	SGI
Alpha 21164	433	12.2	10.5	DEC

- 21 -

CS 740 F99

## Summary

### Alpha

- Simple register state
- Every operation has single effect
  - Load, store, operate, branch

### VAX

- Hidden control state
- Operations vary from simple to complex
- Side effects possible

### Power PC

- Complex control state
- Operations simple to medium
- Side effects possible
- Hard target for code generator

- 22 -

CS 740 F99