

Instruction Set Comparisons CS 740

Sept. 12, 2003

Topics

- **Code Examples**
 - Procedure Linkage
 - Sparse Matrix Code
- **Instruction Sets**
 - Alpha
 - PowerPC
 - VAX
 - x86

Procedure Examples

Procedure linkage

- Passing of control and data
- stack management

Code Example

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

Control

- Register tests
- Condition codes
- loop counters

- 2 -

CS 740 F'03

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

- 3 -

CS 740 F'03

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

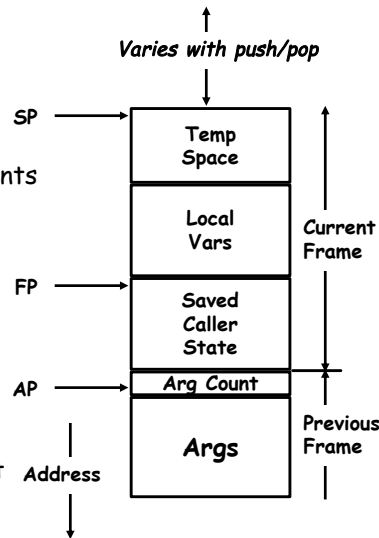
- Direct implementation of stack discipline

- 4 -

CS 740 F'03

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



VAX Operand Specifiers

Forms

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

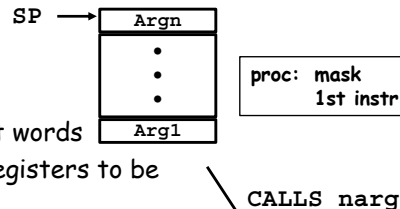
Examples

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

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

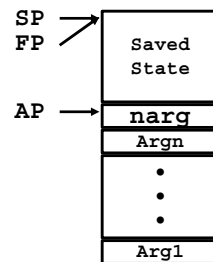


CALLS Instruction

- **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

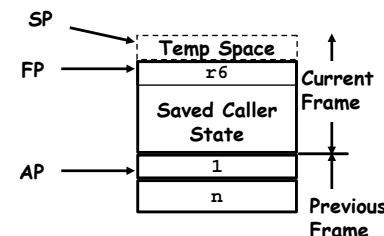
Note

- 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
    
```



x86

- **Pre-history: 8080**
 - 8-bit microprocessor
 - Accumulator machine
- **8086/8088**
 - 16-bit machine
 - Added registers, but many implicit
 - Extended Accumulator Machine
 - Chosen by IBM
 - 16-bit address space
- **8087**
 - Floating point processor
 - Extended stack machine
- **80286**
 - 24-bit address space
- **80386**
 - 32-bit machine
 - Almost a general purpose register machine
 - Still have 8086 compatibility mode
- **80486**
 - MMX
- **P-III**
 - Added SSE
- **P4**
 - Added SSE2
 - Now floating point looks like a general register machine too

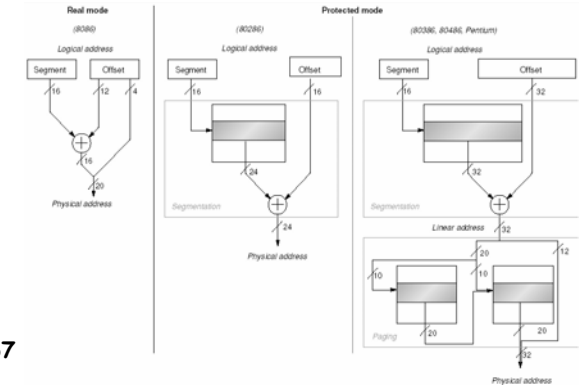
x86 - Memory Architecture

• Segmented architecture

- CS, DS, ES, SS

• Addressing Modes

- Absolute
 $\#0x45612789$
- Register indirect
 r
- Base+displacement
 $r+0x01234567$
- Indexed
 $r+s$
- Indexed+displacement
 $r+s+0x01234567$
- Base + scaled index
 $r+(s \ll \text{scale})$
- Base + disp + scaled index
 $r+0x01234567+(s \ll \text{scale})$



X86 - Registers

- Very few registers. Lots of meaning attached to each
- Accumulator: AX (AH,AL) -> EAX
- Count, loop: CX
- Data, Mult, Div: DX
- Base: BX
- Stack pointer: SP
- Base pointer: BP (fp)
- Source & Dest index regs: SI, DI
- 8 Floating point
- Condition codes, PC

X86 - Instructions

- Standard ones you would expect, and ...
- Call, Callf
- Ret, retf
- Loop: if (--CX) goto PC+0x??
- Push, Pop
- MOVs
- Prefixes: repeat, lock, ...
- Postfixes: address specifiers
- Can vary in length from 1 byte to 17

X86 - rfact

```

_rfact:
  pushl   %ebp                ! save old stack base
  movl   %esp,%ebp          ! point BP to args
  subl   $20,%esp           ! get some more stack space
  pushl   %ebx               ! save reg
  movl   8(%ebp),%ebx        ! Get argument
  cmpl   $1,%ebx
  jle    L3
  leal   -1(%ebx),%eax       ! TRICKY!
  addl   $-12,%esp          ! allocate space for args
  pushl   %eax               ! save arg
  call   _rfact              ! make call
  imull  %eax,%ebx           ! multiply result
  movl   %ebx,%eax          ! put back in accumulator
  jmp    L5
  .align 4
L3:
  movl   $1,%eax
L5:
  movl   -24(%ebp),%ebx      ! restore reg
  movl   %ebp,%esp          ! restore sp
  popl   %ebp               ! restore fp
  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

3.50.9	2.2	[(0,3.5) (1,0.9) (3,2.2)]
4.1	1.9	[(1,4.1) (3,1.9)]
4.6	0.72.7	[(0,4.6) (2,0.7) (3,2.7) (5,3.0)]
	2.9	[(2,2.9)]
1.2	2.8	[(2,1.2) (4,2.8)]
	3.4	[(5,3.4)]

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)]

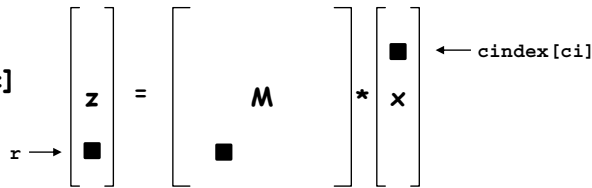
```

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

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

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
    
```

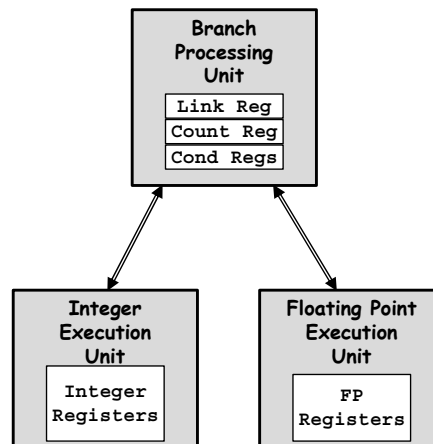
PowerPC Structure

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**



IBM Compiler PPC Inner Loop

Registers

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

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*

```

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
    
```

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

- 25 -

CS 740 F'03

Performance Comparison

Experiment

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

Machine	MHz	µsecs	Cyc/Ele	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

- 26 -

CS 740 F'03

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

X86

- Well, what can you say: best selling desktop/server proc around

- 27 -

CS 740 F'03