Chapter 4
Simple Calculations

Arthur B. Maccabe
Department of Computer Science
The University of New Mexico

Copyright 1993–2000, Arthur B. Maccabe and McGraw-Hill, Inc.

Overview

Slide 1

- Preliminaries
  - RISC/CISC
  - registers, load/store architecture
- SPARC data manipulation instructions
  - registers and data paths
  - arithmetic, logical (bitwise), and shift instructions
  - sethi and synthetic instructions
- SPARC Control Flow
  - labels and unconditional branches
  - conditional branching
  - nullification
  - encoding branching instructions
  - translation issues

Introduction

Slide 2

- Architecture: the design of an instruction set
- Organization: the implementation of an instruction set
- SPARC (Scalable Processor ARChitecture)
  - RISC (Reduced Instruction Set Computer) architecture
  - an open standard
  - Register windows and resource management
- Binary compatibility
- Backward compatibility

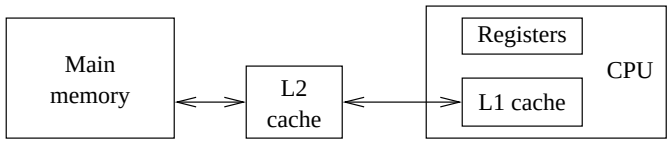
Preliminaries

Slide 3

- RISC/CISC
  - reduced/complex instruction set computer
  - the complexity of individual instructions
  - changing role of assembly language programming
- Kinds of instructions
  - data transfer
  - data mainpulation
  - control flow
- Statements, operations, and operands
  - single operation per instruction
  - operations include store (e.g., multiply-and-store)
  - source operands provide values
  - destination operand provides a location

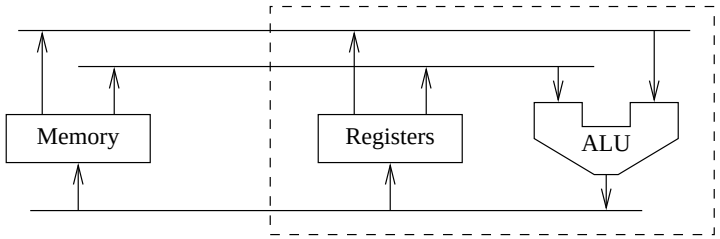
Slide 4

Preliminaries: Registers



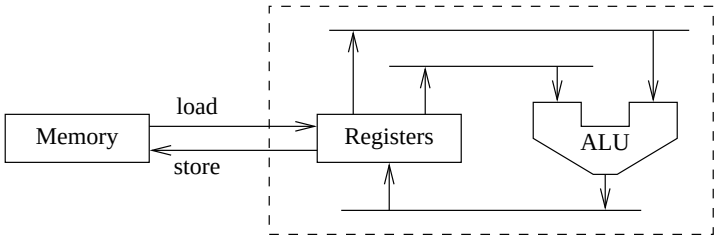
Slide 5

Preliminaries: Memory/Memory Architecture



Preliminaries: Load/Store Architecture

Slide 6



Data Manipulation Instructions

Slide 7

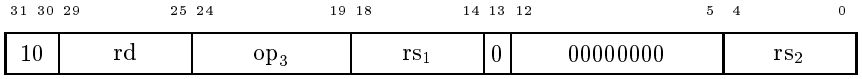
- Background
- Machine language formats
- Internal data paths
- Assembly language
- Arithmetic instructions
- Logical (bitwise) instructions
- Shift and 32-bit shift instructions
- sethi (set high)
- Synthetic instructions
- 32-bit multiply and divide instructions

SPARC Background

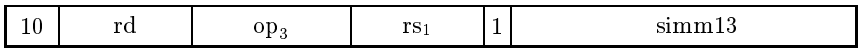
Slide 8

- Basic sizes
  - 32-bit word
  - 64-bit double (or extended) word
- Registers
  - 32 64-bit registers
  - %r0 through %r63
  - %r0 is always zero

SPARC Instruction Formats



Register-register



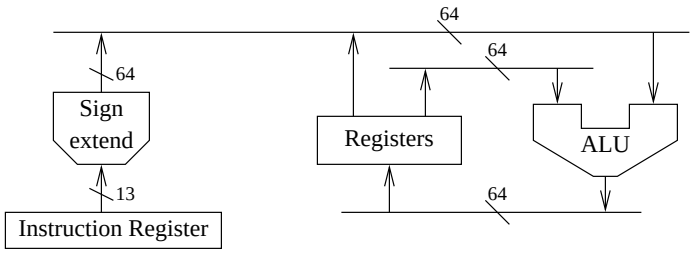
Slide 9

Register-immediate

Field	Meaning
rd	Destination register
op_3	Op code
rs_1	First source register
rs_2	Second source register
simm13	Second source value in 13-bit 2's complement representation

Slide 10

Internal Data Paths



Slide 11

Arithmetic Instructions

Operation	op_3	Assembly syntax	Operation implemented
Integer	00 0000	add rs_1, rs_2, rd	reg[rd] = reg[rs_1] + reg[rs_2]
add		add rs, $siconst_{13}$, rd	reg[rd] = reg[rs] + $siconst_{13}$
Integer	00 0100	sub rs_1, rs_2, rd	reg[rd] = reg[rs_1] − reg[rs_2]
subtract		sub rs, $siconst_{13}$, rd	reg[rd] = reg[rs] − $siconst_{13}$
Integer	00 1001	mulx rs_1, rs_2, rd	reg[rd] = reg[rs_1] * reg[rs_2]
multiply		mulx rs, $siconst_{13}$, rd	reg[rd] = reg[rs] * $siconst_{13}$
Integer	10 1101	sdivx rs_1, rs_2, rd	reg[rd] = reg[rs_1] / reg[rs_2]
divide		sdivx rs, $siconst_{13}$, rd	reg[rd] = reg[rs] / $siconst_{13}$

Note: $siconst_{13}$ denotes a signed integer constant in the range −4096 to 4095, inclusive.

Destination last

Encoding Examples

Slide 12

- encode add1 %r1, 2, %r1

Field	Value	Encoded value
rd	%r1	00001
op_3	mulx	001001
rs_1	%r1	00001
$simm_{13}$	2	0000000000010

10 00001 001001 00001 1 0000000000010 or 0x82486002

- encode add1 %r1, %r10, %r1

Field	Value	Encoded value
rd	%r1	00001
op_3	add	000000
rs_1	%r1	00001
rs_2	%r10	01010

10 00001 000000 00001 0 00000000 01010 or 0x8200400A

Assembly Language Example

Slide 13

- a = a ∗ 2 + b ∗ c ∗ d
  a is in %r1, *b* is in %r2, and *c* is in %r3

```
mulx   %r1, 2, %r1        ! a = a * 2
mulx   %r2, %r3, %r10     ! t1 = b * c
mulx   %r10, %r4, %r10    ! t1 = t1 * d
add    %r1, %r10, %r1     ! a = a + t1
```

- line oriented
- comments start with '!'

Logical (Bitwise) Instructions

Operation	op_3	Assembly syntax	Operation implemented
And	00 0001	and rs_1, rs_2, rd	reg[rd]=reg[rs_1] & reg[rs_2]
		and rs, $siconst_{13}$, rd	reg[rd]=reg[rs] & $siconst_{13}$
And not	00 0101	andn rs_1, rs_2, rd	reg[rd]=reg[rs_1] & ~reg[rs_2]
		andn rs, $siconst_{13}$, rd	reg[rd]=reg[rs] & ~$siconst_{13}$
Inclusive or	00 0010	or rs_1, rs_2, rd	reg[rd]=reg[rs_1] | reg[rs_2]
		or rs, $siconst_{13}$, rd	reg[rd]=reg[rs] | $siconst_{13}$
Inclusive	10 0110	orn rs_1, rs_2, rd	reg[rd]=reg[rs_1] | ~reg[rs_2]
or not		orn rs, $siconst_{13}$, rd	reg[rd]=reg[rs] | ~$siconst_{13}$
Exclusive or	00 0011	or rs_1, rs_2, rd	reg[rd]=reg[rs_1] $\wedge$ reg[rs_2]
		or rs, $siconst_{13}$, rd	reg[rd]=reg[rs] $\wedge$ $siconst_{13}$
Exclusive	10 0111	orn rs_1, rs_2, rd	reg[rd]=reg[rs_1] $\wedge$ ~reg[rs_2]
or not		orn rs, $siconst_{13}$, rd	reg[rd]=reg[rs] $\wedge$ ~$siconst_{13}$

Slide 14

Examples

- Compute the 8-bit, bitwise logical *and* for 11100011 and 10101010

```
    11100011
&   10101010
------------
    10100010
```

- Compute the 8-bit, bitwise logical *or* for 11100011 and 10101010

Slide 15

```
    11100011
|   10101010
------------
    11101011
```

- Clear all but the least significant 10 bits in register *%r4*

```
and   %r4, 0x3ff, %r4
```

- Set all but the least significant 10 bits in register *%r4*

```
orn   %r4, 0x3ff, %r4
```

Shift Instructions

Slide 16

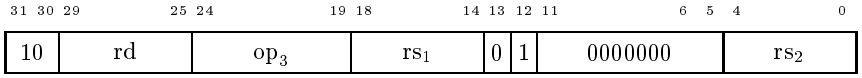
Operation	op_3	Assembler syntax	Operation implemented
left shift	10 0101	sllx rs_1, rs_2, rd	reg[rd] = reg[rs_1] << reg[rs_2]
logical		sllx rs_1, $iconst_6$, rd	reg[rd] = reg[rs_1] << $iconst_6$
right shift	10 0110	srlx rs_1, rs_2, rd	reg[rd] = reg[rs_1] >> reg[rs_2]
logical		srlx rs_1, $iconst_6$, rd	reg[rd] = reg[rs_1] >> $iconst_6$
right shift	10 0111	srax rs_1, rs_2, rd	reg[rd] = reg[rs_1] >> reg[rs_2]
arithmetic		srax rs_1, $iconst_6$, rd	reg[rd] = reg[rs_1] >> $iconst_6$

Note: $iconst_6$ denotes an integer constant in the range 0 to 63, inclusive.

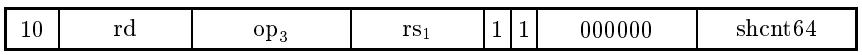
- logical shifts fill vacated bit positions with 0
- arithmetic right shifts replicate the sign bit

Encoding Shift Instructions

Slide 17



Register-register



Register-immediate

Field	Meaning
rd	Destination register
op_3	Op code
rs_1	First source register
rs_2	Second source register
shcnt64	Second source value in 6-bit binary representation

32-bit Shift Instructions

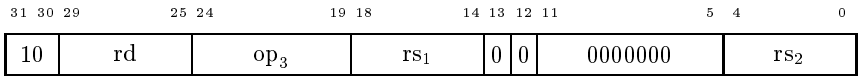
Operation	op_3	Assembler syntax	Operation implemented
left shift	10 0101	sll rs_1, rs_2, rd	reg[rd] = reg[rs_1] << reg[rs_2]
logical		sll rs_1, $iconst_5$, rd	reg[rd] = reg[rs_1] << $iconst_5$
right shift	10 0110	srl rs_1, rs_2, rd	reg[rd] = reg[rs_1] >> reg[rs_2]
logical		srl rs_1, $iconst_5$, rd	reg[rd] = reg[rs_1] >> $iconst_5$
right shift	10 0111	sra rs_1, rs_2, rd	reg[rd] = reg[rs_1] >> reg[rs_2]
arithmetic		sra rs_1, $iconst_5$, rd	reg[rd] = reg[rs_1] >> $iconst_5$

Note: $iconst_5$ denotes an integer constant in the range 0 to 31, inclusive.

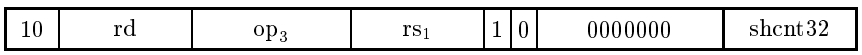
Slide 18

- *sll* is *sllx* with a reduced shift count
- *srl* sets the most significant 32 buts to 0, then acts like *srlx*
- *sra* copies the 32-bit sign bit into most significant 32 bit, then acts like *srax*

Encoding 32-bit Shift Instructions



Register-register



Register-immediate

Slide 19

Field	Meaning
rd	Destination register
op_3	Op code
rs_1	First source register
rs_2	Second source register
shcnt32	Second source value in 5-bit binary representation

sethi: Motivation

Slide 20

using a sequence of shift and or instructions, set the register *%r2* to the 64-bit value 0x1234567887654321.

```
or    %r0, 0x123, %r2   ! start with the most significant 12 bits
sllx  %r2, 12, %r2      ! make room for the next 12 bits
or    %r2, 0x456, %r2   ! the next 12 bits
sllx  %r2, 12, %r2      ! make room for the next 12 bits
or    %r2, 0x788, %r2   ! the next 12 bits
sllx  %r2, 12, %r2      ! make room for the next 12 bits
or    %r2, 0x765, %r2   ! the next 12 bits
sllx  %r2, 12, %r2      ! make room for the next 12 bits
or    %r2, 0x432, %r2   ! the next 12 bits
sllx  %r2, 4, %r2       ! make room for the next 4 bits
or    %r2, 0x1, %r2     ! the last 4 bits
```

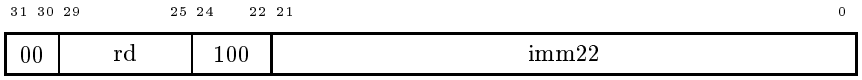
sethi

Slide 21

set the register *%r2* to the 64-bit value 0x1234567887654321.

```
sethi 0x12345, %r2      ! set bits 31 through 12
or    %r2, 0x678, %r2   ! fill in bits 0 through 11
sllx  %r2, 32, %r2      ! shift to the most significant bits
sethi 0x87654, %r3      ! set bits 31 through 12
or    %r3, 0x321, %r3   ! fill in bits 0 through 11
or    %r2, %r3, %r2     ! combine the two 32-bit values
```

Encoding sethi Instructions



Slide 22

Field	Meaning
rd	Destination register
imm22	The 22-bit immediate value

%uhi(), %ulo(), %hi() and %lo()

Slide 23

Operator	Value
%uhi(*val*)	bits 63–42 of *val*
%ulo(*val*)	bits 41–32 of *val*
%hi(*val*)	bits 31–10 of *val*
%lo(*val*)	bits 9–0 of *val*

Using %uhi(), %ulo(), %hi() and %lo()

Slide 24

set the register *%r2* to the 64-bit value 0x1234567887654321.

```
sethi   %uhi(0x1234567887654321), %r2        ! set bits 31 through 12
or      %r2, %ulo(0x1234567887654321), %r2   ! fill in bits 0 through 11
sllx    %r2, 32, %r2                         ! shift to most significant bits
sethi   %hi(0x1234567887654321), %r3         ! set bits 31 through 12
or      %r3, %lo(0x1234567887654321), %r3    ! fill in bits 0 through 11
or      %r2, %r3, %r2                        ! combine the two 32-bit values
```

Synthetic Data Manipulation Instructions

Slide 25

Synthetic instr.	Implementation	Comment
not *rs*, *rd*	xnor *rs*, %r0, *rd*	invert bits
not *rd*	xnor *rd*, %r0, *rd*	
neg *rs*, *rd*	sub %r0, *rs*, *rd*	negation
neg *rd*	sub %r0, *rd*, *rd*	
inc *rd*	add *rd*, 1, *rd*	increment
inc $siconst_{13}$, *rd*	add *rd*, $siconst_{13}$, *rd*	
dec *rd*	sub *rd*, 1, *rd*	decrement
dec $siconst_{13}$, *rd*	sub *rd*, $siconst_{13}$, *rd*	
bset *rs*, *rd*	or *rd*, *rs*, *rd*	set bits
bset $siconst_{13}$, *rd*	or *rd*, $siconst_{13}$, *rd*	

Synthetic Data Manipulation Instructions (continued)

Slide 26

Synthetic instr.	Implementation	Comment
bclr *rs*, *rd*	andn *rd*, *rs*, *rd*	clear bits
bclr $siconst_{13}$, *rd*	andn *rd*, $siconst_{13}$, *rd*	
btog *rs*, *rd*	xor *rd*, *rs*, *rd*	toggle bits
btog $siconst_{13}$, *rd*	xor *rd*, $siconst_{13}$, *rd*	
clr *rd*	or %r0, %r0, *rd*	clear (zero) a register
mov *rs*, *rd*	or %r0, *rs*, *rd*	copy *rs* to *rd*
mov $siconst_{13}$, *rd*	or %r0, $siconst_{13}$, *rd*	set *rd* to $siconst_{13}$

Synthetic Instructions for 32- and 64-bit Values

Slide 27

Synthetic instr.	Implementation	Comment
clruw *rs*, *rd*	srl *rs*, %r0, *rd*	clear upper word
clruw *rd*	srl *rd*, %r0, *rd*	
signx *rs*, *rd*	sra *rs*, %r0, *rd*	sign extend a 32-bit value
signx *rd*	sra *rd*, %r0, *rd*	

The "set unsigned word" Synthetic Instructions

Slide 28

Synthetic instr.	Implementation	Condition
setuw *val*, *rd*	sethi %hi(*val*), *rd*	*val*&0x3ff == 0
set unsigned word	— or —	
	or %r0, *val*, *rd*	$0 \leq val < 4095$
	— or —	
	sethi %hi(*val*), *rd*	otherwise
	or %r0, %lo(*val*), *rd*	
set *val*, *rd*	synonym for setuw	

The "set signed word" Synthetic Instructions

Slide 29

Synthetic instr.	Implementation	Condition
setsw *val*, *rd*	sethi %hi(*val*), *rd*	$0 \leq val$ && *val* & 0x3ff == 0
set signed word	— or —	
	or %r0, *val*, *rd*	$-4096 \leq val < 4095$
	— or —	
	sethi %hi(*val*), *rd*	$val < 0$ && *val* & 0x3ff == 0
	sra *rd*, %r0, *rd*	
	— or —	
	sethi %hi(*val*), *rd*	otherwise, if $val \geq 0$
	or %r0, %lo(*val*), *rd*	
	— or —	
	sethi %hi(*val*), *rd*	otherwise, if $val < 0$
	or %r0, %lo(*val*), *rd*	
	sra *rd*, %r0, *rd*	

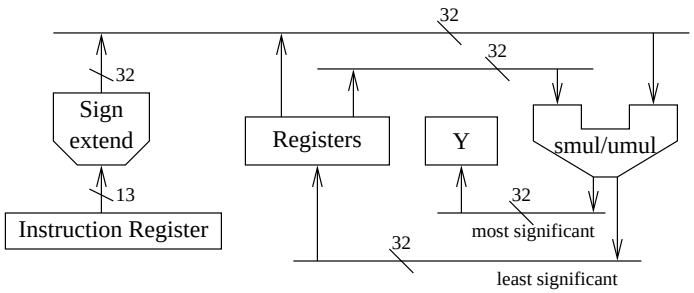
Slide 30

The "set extended word" Synthetic Instructions

Synthetic instr.	Implementation	Condition
setx *val*, *tmp*, *rd*	sethi %uhi(*val*), *tmp*	worst case, optimizations
set extended	or *tmp*, %ulo(*val*), *tmp*	are possible
(*tmp* is a temporary	sllx *tmp*, 32, *tmp*	
register)	sethi %hi(*val*), *rd*	
	or *rd*, *tmp*, *rd*	
	or *rd*, %lo(*val*), *rd*	

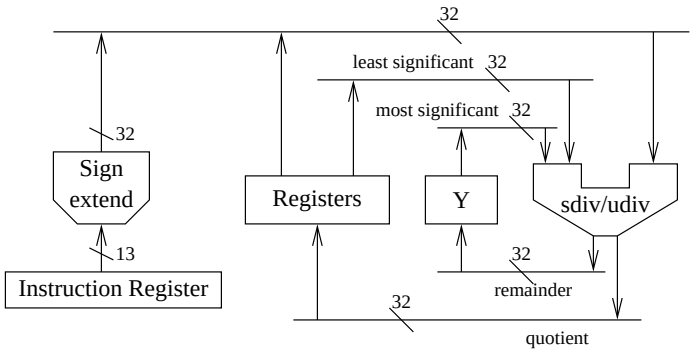
Slide 31

32-bit Multiply



Slide 32

32-bit Divide



Slide 33

Control Transfer Instructions

- conditional and unconditional branches
- labels and unconditional branches on the SPARC
- delayed branches
- register comparison to zero
- condition code registers
- encoding branching instructions
- loop improvements
- complex conditional expressions

An Example

- using a **while** loop

```
temp = y;
x = 0;
while( temp ≥ z ){
    x = x + 1;
    temp = temp − z;
}
```

Slide 34

- using simple **goto**'s

```
    temp = y;
    x = 0;
top:
    if( temp < z ) goto bottom
    x = x + 1;
    temp = temp − z;
  goto top;
bottom:
```

Labels

- A label is an identifier (and follows the standard rules for identifier formation).
- The definition of a label occurs when the label appears as the first non-white space item on a line followed by a colon (":").

Slide 35

- Each label can have at most one definition.
- The value of the label is the address of the next instruction (which may be on the same line as the label).
- Whenever you use a label in a non defining context, the assembler substitutes the value of the label for the label.
- You may use a label before you give its definition.

Delayed Branching

Slide 36

- unconditional branch: ba %xcc, *label*
- *pc* and *npc*
- an example

Code			pc	npc	Comment
s1:	inc	%r1	s1	s2	start
s2:	ba	%xcc, s5	s2	s3	sequential execution
s3:	inc	%r3	s3	s5	(delayed) branch to s5
s4:	inc	%r4			
s5:	inc	%r5	s5	s6	sequential execution
s6:	inc	%r6	s6	s7	
s7:	inc	%r7			

- nop

Translating the Unconditional Branch

Slide 37

x is stored in *%r2*, *y* is stored in *%r3*, *z* is stored in *%r4*, and *temp* is stored in *%r5*

```
        mov   %r3, %r5        ! temp = y;
        clr   %r2             ! x = 0;
top:
                              ! if( temp < z ) goto bottom
        inc   %r2             ! x = x + 1;
        sub   %r5, %r4, %r5   ! temp = temp - z;
        ba    %xcc, top       ! goto top;
        nop                   ! delay slot
bottom:
```

Eliminating the *nop*

Slide 38

```
            mov   %r3, %r5        ! temp = y;
            clr   %r2             ! x = 0;
      top:
                                  ! if( temp < z ) goto bottom
            inc   %r2             ! x = x + 1;
            ba    %xcc, top       ! goto top;
            sub   %r5, %r4, %r5   ! temp = temp - z; –delay slot–
   bottom:
```

Conditional Branching

Slide 39

- register comparison to zero
- condition codes

Slide 40

Register Comparison to Zero

Assembler syntax	Branching condition
brz *rs*, *label*	reg[*rs*] = 0
brlez *rs*, *label*	reg[*rs*] $\leq$ 0
brlz *rs*, *label*	reg[*rs*] $<$ 0
brnz *rs*, *label*	reg[*rs*] $\neq$ 0
brgz *rs*, *label*	reg[*rs*] $>$ 0
brgez *rs*, *label*	reg[*rs*] $\geq$ 0

Slide 41

Using Comparisons to Zero

- simple translation

```
    temp = y;
    x = 0;
top:
    temp2 = temp − z;
    if( temp2 < 0 ) goto bottom
        x = x + 1;
        temp = temp − z;
    goto top;
bottom:
```

Using Comparisons to Zero (continued)

Slide 42

- avoiding *temp2*

```
    temp = y − z;
    x = 0;
top:
    if( temp < 0 ) goto bottom
        x = x + 1;
        temp = temp − z;
    goto top;
bottom:
```

Translating into SPARC Code

Slide 43

```
        sub   %r3, %r4, %r5   ! temp = y − z;
        clr   %r2             ! x = 0;
top:
        brlz  %r5, bottom     ! if( temp < 0 ) goto bottom
        nop                   ! branch delay slot
        inc   %r2             ! x = x + 1;
        ba    %xcc, top       ! goto top;
        sub   %r5, %r4, %r5   ! temp = temp - z; –delay slot–
bottom:
```

Avoiding the *nop* Instruction

Slide 44

```
        sub   %r3, %r4, %r5   ! temp = y - z;
        clr   %r2             ! x = 0;
top:
        brlz  %r5, bottom     ! if( temp < 0 ) goto bottom
        inc   %r2             ! x = x + 1; -delay slot-
        ba    %xcc, top       ! goto top;
        sub   %r5, %r4, %r5   ! temp = temp - z; -delay slot-
bottom:
        dec   %r2             ! make up for the extra inc
```

Condition Codes

Slide 45

- example
- condition code bits (flangs)
- setting the condition codes
- conditional branching based on condition codes

Another Example

Slide 46

- the code

```
if( a > b ) {
    a = a + 1;
}
```

- first translation

```
c = a − b;
if( c ≤ 0 ) goto fi;
    a = a + 1;
fi:
```

- translation to SPARC code

```
        sub     %r2, %r3, %r4   ! c = a − b;
        brlez   %r4, fi         ! if( c ≤ 0 ) goto fi;
        nop                     ! branch delay slot
        inc     %r2             ! a = a + 1;
fi:
```

Condition Code Bits (Flags)

Slide 47

- summarize the result of an earlier operation
- Z, zero
- N, negative
- C, carry
- V, overflow

Setting the Condition Code Bits

Slide 48

	Operations that update the condition codes		Operations that do not affect the condition codes	
Operation	Name	op_3	Name	op_3
integer addition	addcc	01 0000	add	00 0000
integer subtraction	subcc	01 0100	sub	00 0100
bitwise and	andcc	01 0001	and	00 0001
bitwise and not	andncc	01 0101	andn	00 0101
bitwise or	orcc	01 0010	or	00 0010
bitwise or not	orncc	01 0110	orn	00 0110
bitwise xor	xorcc	01 0011	xor	00 0011
bitwise xor not	xorncc	01 0111	xorn	00 0111

Synthetic Instructions that Update the Condition Code

Slide 49

Synthetic instr.	Implementation	Comment
cmp rs_1, rs_2	subcc rs_1, rs_2, %r0	compare
cmp *rs*, $siconst_{13}$	subcc *rs*, $siconst_{13}$, %r0	
tst *rs*	orcc %r0, *rs*, %r0	test
inccc *rd*	addcc *rd*, 1, *rd*	increment
inccc $siconst_{13}$, *rd*	addcc *rd*, $siconst_{13}$, *rd*	
deccc *rd*	subcc *rd*, 1, *rd*	decrement
deccc $siconst_{13}$, *rd*	subcc *rd*, $siconst_{13}$, *rd*	
btst *rs*, *rd*	andcc *rd*, *rs*, *rd*	test bits
btst $siconst_{13}$, *rd*	andcc *rd*, $siconst_{13}$, *rd*	

The Condition Codes

Slide 50

- two condition codes are updated
- %icc – 32-bit result
- %xcc – 64-bit result

Branching Based on the Condition Codes

Slide 51

Name	Condition	Comment
ba	*always*	unconditional branch always
bn	*never*	unconditional branch never
bne	$\overline{Z}$	not equal (to zero)
bnz	$\overline{Z}$	nonzero (a synonym for bne)
be	Z	equal (to zero)
bz	Z	zero (a synonym for be)
bg	$\overline{Z \text{ or } (N \text{ xor } V)}$	greater (than zero)
ble	$Z \text{ or } (N \text{ xor } V)$	Less or equal (to zero)
bge	$\overline{N \text{ xor } V}$	greater or equal (to zero)
bl	$N \text{ xor } V$	less (than zero)

Branching Based on the Condition Codes (continued)

Slide 52

Name	Condition	Comment
bgu	$\overline{C \text{ or } Z}$	greater (than zero), unsigned
bleu	C or Z	less or equal (to zero), unsigned
bcc	$\overline{C}$	carry clear
bgeu	$\overline{C}$	greater or equal (to zero), unsigned (synonym for bcc)
bcs	C	carry set
blu	C	less (than zero), unsigned
bpos	$\overline{N}$	positive
bneg	N	negative
bvc	$\overline{V}$	overflow clear
bvs	V	overflow set

SPARC Code Examples

Slide 53

- basic translation

```
        sub   %r2, %r3, %r0   ! a − b;
        ble   %xcc, fi        ! if( c ≤ 0 ) goto fi;
        nop                   ! branch delay slot
        inc   %r2             ! a = a + 1;
fi:
```

- using a *cmp* synthetic instruction

```
        cmp   %r2, %r3
        ble   %xcc, fi    ! if( a ≤ b ) goto fi;
        nop               ! branch delay slot
        inc   %r2         ! a = a + 1;
fi:
```

Nullification: Motivation

Slide 54

- cancelling the effect of an instruction in the branch delay slot
- example
  - original

```
        cmp  %r2, %r3
        ble  %xcc, fi    ! if( a ≤ b ) goto fi;
        nop              ! branch delay slot
        inc  %r2         ! a = a + 1;
    fi:
```

  - without using a *nop* instruction

```
        cmp  %r2, %r3
        bg   %xcc, fi    ! if( a > b ) goto fi;
        inc  %r2         ! a = a + 1; –delay slot–
        dec  %r2         ! a = a − 1; cancel
    fi:
```

Nullification

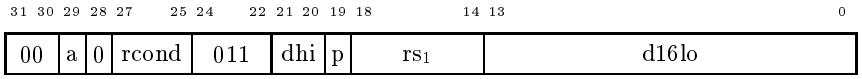
Slide 55

- for conditional branching instructions:
  - branch taken: instruction in delay slot is executed
  - branch not taken: affect of instruction in delay slot can be nullified
- for unconditional branching (*ba* and *bn*): instruction in delay slot can be nullified
- nullification is specified by concatenating ",a" to the operation name
- SPARC uses the name annul instead of nullify
- example

```
        cmp  %r2, %r3
        bg,a %xcc, fi    ! if( a > b ) goto fi;
        inc  %r2         ! a = a + 1; –delay slot–
    fi:
```

Encoding Branches Using Register Comparison to Zero

Slide 56

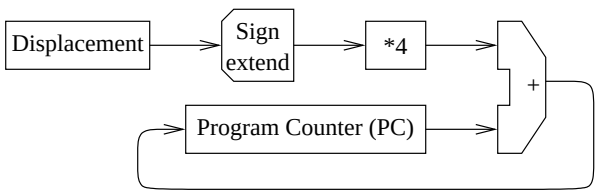


Field	Meaning
a	Annul (nullify): 1 to annul the instruction in the delay slot
rcond	Branching condition
dhi	Most significant 2 bits of the displacement
p	Prediction: 1 to predict branch taken, 0 to predict branch *not taken*
rs_1	Source register
d16lo	Least significant 14 bits of the displacement

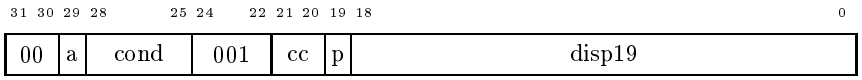
Name	rcond	Name	rcond
brz	001	brnz	101
brlez	010	brgz	110
brlz	011	brgez	111

Displacement Addressing

Slide 57



Encoding Branches Based on the Condition Codes



Slide 58

Field	Meaning
a	Annul (nullify): 1 to annul the instruction in the delay slot
cond	Branching condition
cc	Condition code used in branching: 00 for %icc, 10 for %xcc
p	Prediction: 1 to predict branch taken, 0 to predict branch *not taken*
disp19	19-bit displacement

Encoding the Condition

Slide 59

Name	cond	Name	cond
bn	0000	ba	1000
be, bz	0001	bne, bnz	1001
ble	0010	bg	1010
bl	0011	bge	1011
bleu	0100	bgu	1100
bcs, blu	0101	bcc, bgeu	1101
bneg	0110	bpos	1110
bvs	0111	bvc	1111

Instruction Encoding Examples

Slide 60

- code

```
top:
        brlz    %r5, bot        ! if( temp < 0 ) goto bot
        inc     %r2             ! x = x + 1; -delay-
        ba      %xcc, top       ! goto top;
        sub     %r5, %r4, %r5   ! temp = temp - z; -delay-
bot:
```

Instruction Encoding Examples (Continued)

Slide 61

- encoding the *brlz* instruction

Field	Encoding
a	0
rcond	011
dhi	00
p	1
rs_1	00101
d16lo	00000000000011

00 0 0 011 011 00 1 00101 00000000000011 or 0x06c94003.

Instruction Encoding Examples (Continued)

Slide 62

- encoding the *ba* instruction

Field	Encoding
a	0
cond	1000
cc	10
p	1
disp19	1111111111111111110

00 0 1000 001 10 1 1111111111111111110 or 0x106ffffe.

Translating Loops

Slide 63

- translate to code that uses simple conditional unconditional branches
- try to eliminate unconditional branches in loops
- translate to SPARC assembly code, using *nop* instructions in the branch delay slots
- put *real* instructions in the branch delay slots

Eliminate Unconditional Branches in Loops

Slide 64

- move the test to the bottom of the loop
- example

```
    temp = y − z;
    x = 0;
    goto test;
top:
        x = x + 1;
        temp = temp − z;
test:
    if( temp ≥ 0 ) goto top;
```

Translate to SPARC Code

Slide 65

```
        sub   %r3, %r4, %r5   ! temp = y − z;
        clr   %r2             ! x = 0;
        ba    %xcc, test      ! goto test;
        nop                   ! branch delay slot
top:
        inc   %r2             ! x = x + 1;
        sub   %r5, %r4, %r5   ! temp = temp - z;
test:
        brge  %r5, top        ! if( temp ≥ 0 ) goto top
        nop                   ! branch delay slot
```

Put *Real* Instructions in the Branch Delay Slots

Slide 66

```
        sub     %r3, %r4, %r5   ! temp = y - z;
        ba      %xcc, test      ! goto test;
        clr     %r2             ! x = 0; –delay slot–
top:
        sub     %r5, %r4, %r5   ! temp = temp - z;
test:
        brge,a  %r5, top        ! if( temp ≥ 0 ) goto top
        inc     %r2             ! x = x + 1; –delay slot–
```

More Complex Conditional Expressions

Slide 67

- example

```
if( 20 < a && a < 200 ) {
    b = 100;
} else {
    b = −100;
}
```

- translation

```
if( 20 ≥ a ) goto fals1;
if( a ≥ 200 ) goto fals1;
    b = 100;
    goto fi1;
fals1:
    b = −100;
fi1:
```

Slide 68

Summary