# Chapter 4
# Simple Calculations

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



## 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

Slide 2

## Introduction

- 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

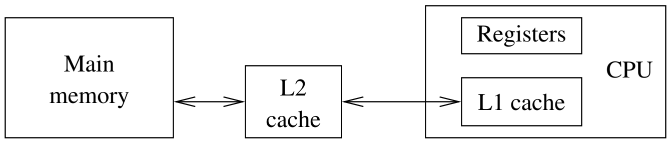
Slide 3

## Preliminaries

- 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

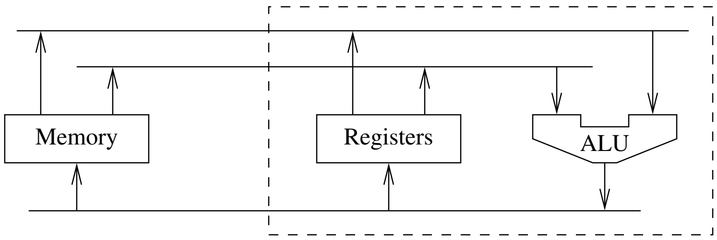
## Preliminaries: Registers

**Slide 4**



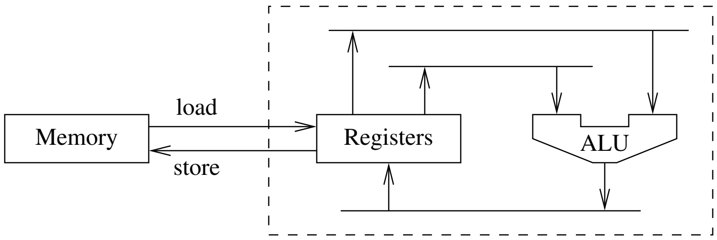

## Preliminaries: Memory/Memory Architecture

**Slide 5**



## 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

**Slide 8**

## SPARC Background

- Basic sizes
  - 32-bit word
  - 64-bit double (or extended) word
- Registers
  - 32 64-bit registers
  - %r0 through %r31
  - %r0 is always zero

**Slide 9**

## SPARC Instruction Formats

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 5 | 4 0 |
|---|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 0 | 00000000 | $rs_2$ |

Register-register

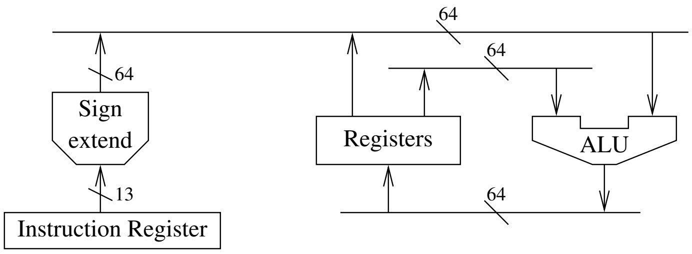
| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 0 |
|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 1 | simm13 |

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 |

## Internal Data Paths

**Slide 10**




## Arithmetic Instructions

**Slide 11**

| 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 Example 1

Slide 12

- SPARC instruction: mulx %r1, 2, %r1
- Format

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 0 |
|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 1 | simm13 |

- Field values

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

- Encoding: 10 00001 001001 00001 1 0000000000010 or 0x82486002

## Encoding Example 2

Slide 13

- SPARC instruction: add %r1, %r10, %r1
- Format

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 5 | 4 0 |
|---|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 0 | 00000000 | $rs_2$ |

- Field values

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

- Encoding: 10 00001 000000 00001 0 00000000 01010 or 0x8200400A

## Assembly Language Example

**Slide 14**

- $a = a * 2 + b * c * d$
  *a* is in %r1, *b* is in %r2, *c* is in %r3, and *d* is in %r4

```
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

**Slide 15**

| 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 | 00 0010 | or $rs_1, rs_2, rd$ | reg[$rd$]=reg[$rs_1$]\| reg[$rs_2$] |
| or | | 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 | 00 0011 | xor $rs_1, rs_2, rd$ | reg[$rd$]=reg[$rs_1$]ˆreg[$rs_2$] |
| xor | | or $rs, siconst_{13}, rd$ | reg[$rd$]=reg[$rs$]ˆ$siconst_{13}$ |
| Exclusive | 10 0111 | xorn $rs_1, rs_2, rd$ | reg[$rd$]=reg[$rs_1$]ˆ∼reg[$rs_2$] |
| xor not | | orn $rs, siconst_{13}, rd$ | reg[$rd$]=reg[$rs$]ˆ∼$siconst_{13}$ |

## Examples

Slide 16

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

$$\begin{array}{rr} & 11100011 \\ \& & 10101010 \\ \hline & 10100010 \end{array}$$

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

$$\begin{array}{rr} & 11100011 \\ | & 10101010 \\ \hline & 11101011 \end{array}$$

- 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 17

| Operation | $op_3$ | Assembler syntax | Operation implemented |
|---|---|---|---|
| left shift | 10 0101 | sllx $rs_1$, $rs_2$, $rd$ | $reg[rd] = reg[rs_1] \ll reg[rs_2]$ |
| logical | | sllx $rs_1$, $iconst_6$, $rd$ | $reg[rd] = reg[rs_1] \ll iconst_6$ |
| right shift | 10 0110 | srlx $rs_1$, $rs_2$, $rd$ | $reg[rd] = reg[rs_1] \gg reg[rs_2]$ |
| logical | | srlx $rs_1$, $iconst_6$, $rd$ | $reg[rd] = reg[rs_1] \gg iconst_6$ |
| right shift | 10 0111 | srax $rs_1$, $rs_2$, $rd$ | $reg[rd] = reg[rs_1] \gg reg[rs_2]$ |
| arithmetic | | srax $rs_1$, $iconst_6$, $rd$ | $reg[rd] = reg[rs_1] \gg iconst_6$ |

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

**Slide 18**

## Encoding Shift Instructions

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 | 11 6 5 | 4 0 |
|---|---|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 0 | 1 | 0000000 | $rs_2$ |

Register-register

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 | 11 6 | 5 0 |
|---|---|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 1 | 1 | 000000 | shcnt64 |

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 |

**Slide 19**

## 32-bit Shift Instructions

| Operation | $op_3$ | Assembler syntax | Operation implemented |
|---|---|---|---|
| left shift | 10 0101 | sll $rs_1, rs_2, rd$ | $\text{reg}[rd] = \text{reg}[rs_1] \ll \text{reg}[rs_2]$ |
| logical | | sll $rs_1, iconst_5, rd$ | $\text{reg}[rd] = \text{reg}[rs_1] \ll iconst_5$ |
| right shift | 10 0110 | srl $rs_1, rs_2, rd$ | $\text{reg}[rd] = \text{reg}[rs_1] \gg \text{reg}[rs_2]$ |
| logical | | srl $rs_1, iconst_5, rd$ | $\text{reg}[rd] = \text{reg}[rs_1] \gg iconst_5$ |
| right shift | 10 0111 | sra $rs_1, rs_2, rd$ | $\text{reg}[rd] = \text{reg}[rs_1] \gg \text{reg}[rs_2]$ |
| arithmetic | | sra $rs_1, iconst_5, rd$ | $\text{reg}[rd] = \text{reg}[rs_1] \gg iconst_5$ |

Note: $\text{iconst}_5$ denotes an integer constant in the range 0 to 31, inclusive.

- *sll* is *sllx* with a reduced shift count
- *srl* sets the most significant 32 bits 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

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 | 11 5 | 4 0 |
|---|---|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 0 | 0 | 0000000 | $rs_2$ |

Register-register

| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 | 11 5 | 4 0 |
|---|---|---|---|---|---|---|---|
| 10 | rd | $op_3$ | $rs_1$ | 1 | 0 | 0000000 | shcnt32 |

**Slide 20**

Register-immediate

| 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

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

**Slide 21**

```
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 22**

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 23**

| 31 30 | 29 25 | 24 22 | 21 0 |
|---|---|---|---|
| 00 | rd | 100 | imm22 |

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

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

**Slide 24**

| 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 25**

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 signif. 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 two 32-bit values
```

## Synthetic Data Manipulation Instructions

**Slide 26**

| 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 27**

| 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 28**

| 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 29**

| 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 30**

| 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* | |

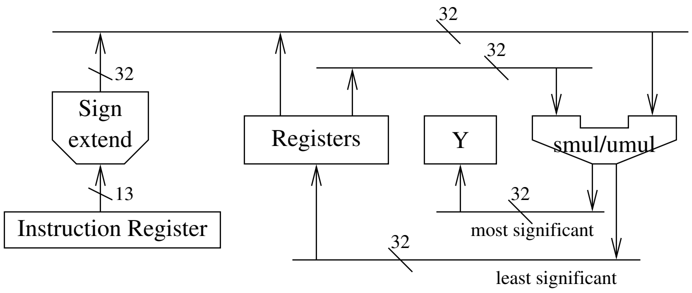
## The "set extended word" Synthetic Instructions

**Slide 31**

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

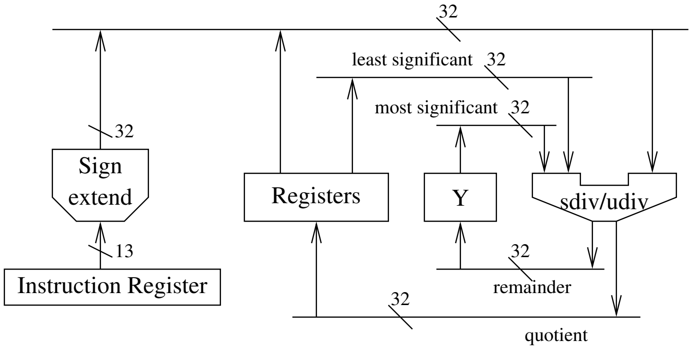
## 32-bit Multiply

**Slide 32**




## 32-bit Divide

**Slide 33**



## Control Transfer Instructions

**Slide 34**

- 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

**Slide 35**

- Using a **while** loop

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

- Using simple **goto**'s

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

## Labels

**Slide 36**

- 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 (":").
- 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 37**

- Unconditional branch: ba %xcc, *label*
- Two program counters: *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 | | | |

- The nop instruction

## Translating the Unconditional Branch

**Slide 38**

Assume *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 39**

```
        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 40**

- Register comparison to zero
- Condition codes

## Register Comparison to Zero

**Slide 41**

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

## Using Comparisons to Zero

**Slide 42**

- 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 43**

- 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 44**

```
        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 45**

```
        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 46**

- Example
- Condition code bits (flags)
- Setting the condition codes
- Conditional branching based on condition codes

## Another Example

**Slide 47**

- 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 48**

- Summarize the result of an earlier operation
- Four flags:
  - Z, zero
  - N, negative
  - C, carry
  - V, overflow

## Setting the Condition Code Bits

**Slide 49**

| | 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 50**

| 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 51**

- Two condition codes are updated
- The 32-bit result: %icc
- The 64-bit result: %xcc

## Branching Based on the Condition Codes

**Slide 52**

| 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 53**

| Name | Condition | Comment |
|---|---|---|
| bgu | $\overline{C \text{ or } Z}$ | greater (than zero), unsigned |
| bleu | $C \text{ or } Z$ | 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 54**

- 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 55**

- 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 56

- 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
- 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 57

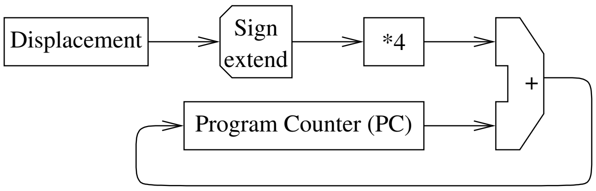
| 31 30 | 29 | 28 | 27 25 | 24 22 | 21 20 | 19 | 18 14 | 13 0 |
|---|---|---|---|---|---|---|---|---|
| 00 | a | 0 | rcond | 011 | dhi | p | $rs_1$ | d16lo |

| 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 58**




## Encoding Branches Based on the Condition Codes

| 31 30 | 29 | 28 25 | 24 22 | 21 20 | 19 | 18 0 |
|---|---|---|---|---|---|---|
| 00 | a | cond | 001 | cc | p | disp19 |

**Slide 59**

| 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 60

| 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 61

- 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 62**

- Encoding the *brlz* instruction
- Format

| 31 30 | 29 | 28 | 27 25 | 24 22 | 21 20 | 19 | 18 14 | 13 0 |
|---|---|---|---|---|---|---|---|---|
| 00 | a | 0 | rcond | 011 | dhi | p | $rs_1$ | d16lo |

- Field values

| Field | Encoding |
|---:|---|
| a | 0 |
| rcond | 011 |
| dhi | 00 |
| p | 1 |
| $rs_1$ | 00101 |
| d16lo | 00000000000011 |

- Encoding: 00 0 0 011 011 00 1 00101 00000000000011 or 0x06c94003.

## Instruction Encoding Examples (Continued)

**Slide 63**

- Encoding the *ba* instruction
- Format

| 31 30 | 29 | 28 25 | 24 22 | 21 20 | 19 | 18 0 |
|---|---|---|---|---|---|---|
| 00 | a | cond | 001 | cc | p | disp19 |

- Field values

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

- Encoding: 00 0 1000 001 10 1 1111111111111111110 or 0x106ffffe.

## Translating Loops

**Slide 64**

- 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 65**

- 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 66**

```
        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 67**

```
        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 68**

- 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:
```