SPARC V8 Architecture

Overview

```
4          or %r0, 40, %r1,          //Set R1 to operands address
8          ldub [%r1,%r0], %r2       //Load multiplicand
12         ldub [%r1, 2], %r3        //Load multiplier
16         add %r0, %r0, %r5         //Clear R5
20   Loop  add %r2, %r5, %r5         //Add R2 and R5 (cumulative)
24         subcc %r3, 1, %r3         //Subtract multiplier
28         bne Loop                  //Repeat iteration
32         stb %r5, [%r1,3]          //Store result
36         ba End                    //Branch to Loc 44
40
44   End   ba End                    //Endless loop
```

Organization of the Lesson

Part I: Overview

Part II: Arithmetic, Logical and Shift Instructions

Part III: Load/Store Instructions

Part IV: Branch Instructions

Part V: Subroutine Instructions

Part VI: Trap Instructions

Part VII: Read/Write State Register Instructions

Part VIII: Coding and Decoding Instructions

SPARC Architecture Versions

SPARC (**S**calable **P**rocessor **ARC**hitecture) developed by Sun Microsystems in 1985.

- **SPARC V7**: first version, 32-bit architecture released in 1986
- **SPARC V8**: first version, 32-bit architecture released in 1990
- **SPARC V9**: 64-bit architecture released in 1993
- **UltraSPARC Architecture 2005**: released in 2005
- **Oracle SPARC Architecture 2015**: released in 2015

Registers

- General Purpose Registers (thirty two 32-bit registers)
- Floating Point Registers (thirty two 32-bit registers)
- **PC** and **nPC**
- **CWP** (Current Window Pointer)
- **WIM** (Window Invalid Mask)
- **TBR** (Trap Base Register)
- **PSR** (Processor State Register)
- Ancillary State Registers

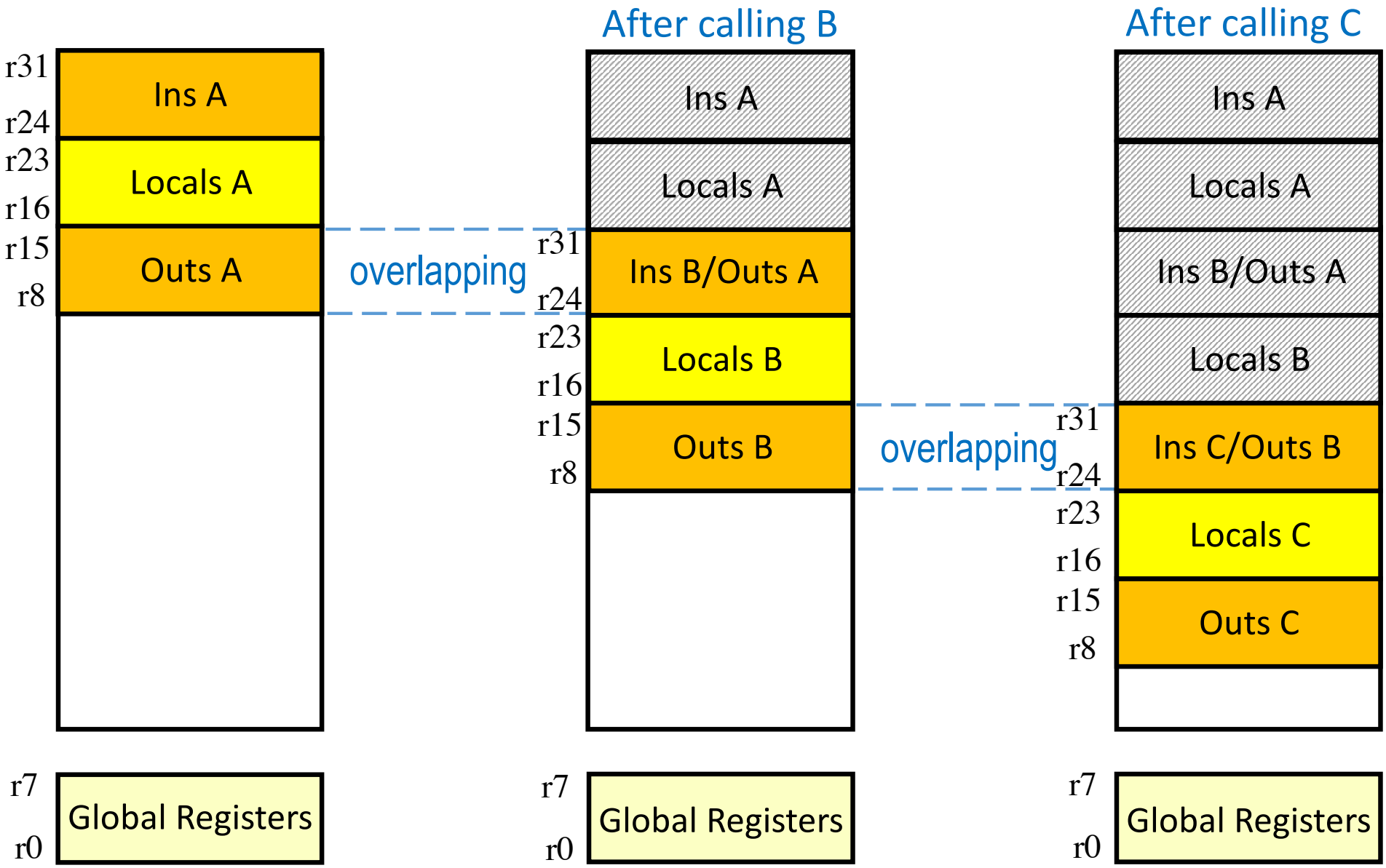
General Purpose Registers

- Thirty two 32-bit registers (r0-r31)
- Global Registers g0-g7 (r0-r7)
  - Are always visible
  - g0 (r0) has always a value of zero (writing it has no effect)
- Windowed Registers
  - Registers r8-r31 are associated with a Register Windows mechanism used for supporting subroutines and traps

Register Windows

- The Register Windows mechanisms consists of blocks of 24 registers (a window)
- Only one block can be visible during the execution of a subroutine
- On subroutine calls and returns, traps and returns from traps a new block of registers is made visible
- The registers of a window are divided in three groups:
  - In registers: i0-i7 (r24-r31)
  - Local registers: l0-l7 (r16-r23)
  - Out registers: o0-07 (r8-r15)
- The Out registers of a subroutine overlaps with the In registers of a subroutine called by it and vice versa

Overlapping of Registers by Subroutine Calls

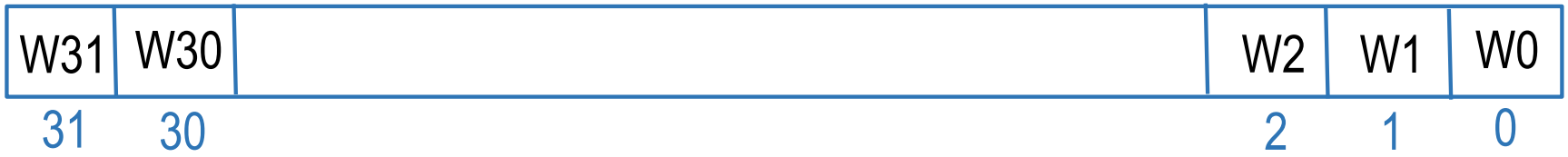


Registers for Handling Register Windows Mechanism

CWP (Current Window Pointer)

- Indicates the active window
- Specified by the least significant 5 bits of **PSR**

WIM (Window Invalid Mask)



- A 1 on a given bit of **WIM** indicates an invalid window (has data that belongs to a previous subroutine)

The number of windows implemented must be 2n where n cannot be greater than 5 (a maximum of 32 windows).

PC and nPC Registers

PC (Program Counter)

- 32-bit register that contains the address of the instruction currently being executed
- Least significant two bits are always zero

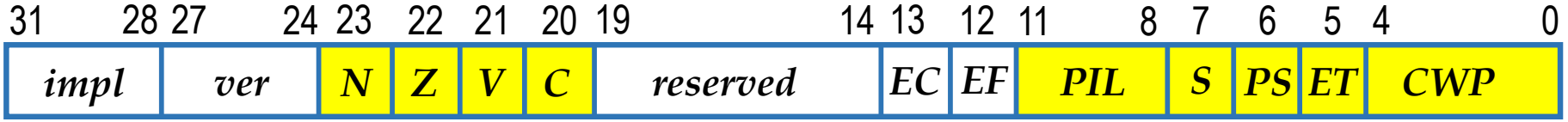
NPC (Next Program Counter)

- 32-bit register that contains the address of the next instruction to be executed if a trap does not occur.
- Least significant two bits are always zero

TBR (Trap Base Register)

- Holds the address of the first instruction of a trap routine

Processor State Register (PSR)



impl – identify implementation or class of implementation

ver – implementation-dependent

N- negative condition code

Z- zero condition code

V- overflow condition code

C- Carry condition code

EC- enable co-processor

EF- enable floating point unit

PIL- Processor Interrupt Level

S- processor operation mode

1= supervisor mode
0 = user mode

PS – Value of ***S*** at the time of the most recent trap

ET- Trap enable

1= traps enabled
0= traps disabled

CWP- Current Window Pointer

Ancillary State Registers

- 31 implementation-dependent registers
- Read and written with the RDASR and WRASR instructions

Memory

- 2^{32} bytes
- Instructions must begin on even addresses, multiples of four
- Big endian format
- Access to I/O is via DMA (direct memory access)

Data Types

- Integer
  - Byte
  - Halfword (16 bits)
  - Word (32 bits)
  - Doubleworld (64 bits, two words)
- Floating point

Types of Instructions

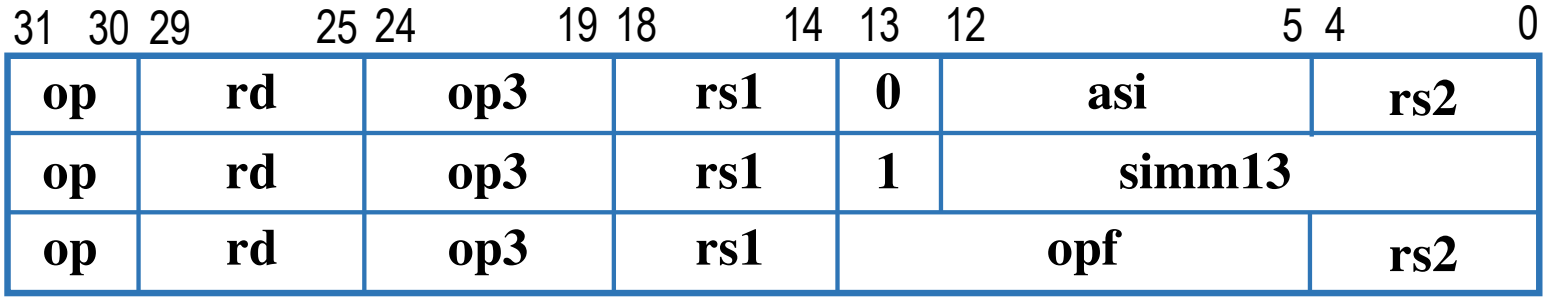
- Arithmetic/logical/shift
- Load/store
- Control transfer
- Read/write control register
- Floating-point operate
- Coprocessor operate

Instruction Formats

Format 1 (*op* = 01): CALL

Format 2 (*op* = 00): SETHI & Branches (Bicc, FBfcc, CBccc)

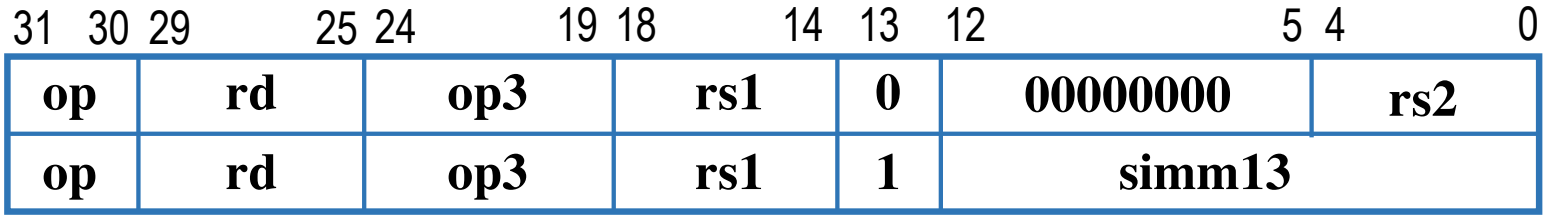
Format 3 (*op* = 10 or 11): Remaining instructions



SPARC V8 Architecture

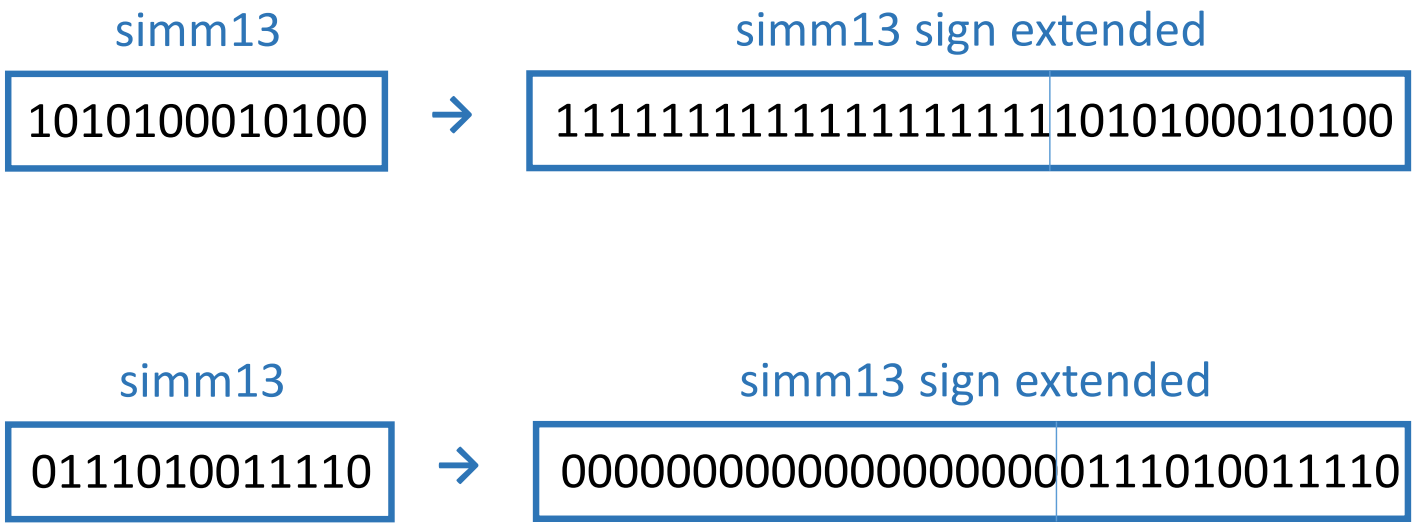
Arithmetic, Logical and Shift Instructions

Instruction Formats



- op = 10
- Two modalities
  - If bit 13 = 0, **rd** <= **rs1** operation **rs2**
  - If bit 13 = 1, **rd** <= **rs1** operation **simm13** *(sign extended)*

simm13 Sign Extension



Basic Arithmetic Instructions

Instruction	op3	Operation
add	000000	rd <= rs1 + (rs2 or simm13)
addcc	010000	rd <= rs1 + (rs2 or simm13), modify icc
addx	001000	rd <= rs1 + (rs2 or simm13) + Carry
addxcc	011000	rd <= rs1 + (rs2 or simm13) + Carry, modify icc
sub	000100	rd <= rs1 - (rs2 or simm13)
subcc	010100	rd <= rs1 - (rs2 or simm13), modify icc
subx	001100	rd <= rs1 - (rs2 or simm13) - Carry
subxcc	011100	rd <= rs1 - (rs2 or simm13) - Carry, modify icc

Condition Codes

- **Z** is set to 1 if the result of an operation is zero, otherwise is set to 0.
- **N** is set to 1 if the most significant bit of the result is a 1, otherwise is set to a 0.
- **C** is set to 1 is the result of the addition or a subtraction requires 33 bits to be correctly represented.
- For addition instructions **V** is set to 1 when the source operands have the same sign and the result has the opposite sign, otherwise it is set to 0.
- For subtraction instructions **V** is set to 1 when the source operands have different signs and the result has the opposite sign of rs1, otherwise it is set to 0.

Tagged Arithmetic Instructions

Instruction	op3	Operation
taddcc	100000	Tagged Add and modify icc
taddcctv	100010	Tagged Add, modify icc and trap on overflow
tsubcc	100001	Tagged Subtract and modify icc
tsubcctv	100011	Tagged Subtract, modify icc and trap on overflow

- *The main difference between tagged add and subtract instructions from from their basic counterparts is the way the overflow condition bit is determined.*
- *For these instructions the overflow bit is set if an arithmetic overflow takes place or bit 0 or bit 1 of any of the source operands are non zero.*
- *If a taddcctv or a tsubcctv instruction causes a tag_overflow, a tag_overflow trap is generated rd and the condition codes remain unchanged.*

Other Arithmetic Instructions

Instruction	op3	Operation
mulscc	100101	Multiply Step and modify icc
umul	001010	Unsigned Integer Multiply
umulcc	011010	Unsigned Integer Multiply and modify icc
smul	001001	Signed Integer Multiply
smulcc	011001	Signed Integer Multiply and modify icc
udiv	001110	Unsigned Integer Divide
udivcc	011110	Unsigned Integer Divide and modify icc
sdiv	001111	Signed Integer Divide
sdivcc	011111	Signed Integer Divide and modify icc

Logical Instructions

Instruction	op3	Operation
and	000001	rd <= rs1 bitwise AND (rs2 or simm13)
andcc	010001	rd <= rs1 bitwise AND (rs2 or simm13), modify icc
andn	000101	rd <= rs1 bitwise AND (NOT(rs2 or simm13))
andncc	010101	rd <= rs1 bitwise AND (NOT(rs2 or simm13)), modify icc
or	000010	rd <= rs1 bitwise OR (rs2 or simm13)
orcc	010010	rd <= rs1 bitwise OR (rs2 or simm13), modify icc
orn	000110	rd <= rs1 bitwise OR (NOT(rs2 or simm13))
orncc	010110	rd <= rs1 bitwise OR (NOT(rs2 or simm13)), modify icc
xor	000011	rd <= rs1 bitwise XOR (rs2 or simm13)
xorcc	010011	rd <= rs1 bitwise XOR (rs2 or simm13), modify icc
xorn	000111	rd <= rs1 bitwise XNOR (rs2 or simm13)
xorncc	010111	rd <= rs1 bitwise XNOR (rs2 or simm13), modify icc

Shift Instructions

Instruction	op3	Operation
sll	100101	shift left logical rs1 *count* positions
srl	100110	shift right logical rs1 *count* positions
sra	100111	shift right arithmetic rs1 *count* positions rd = MSB of rs1)

- *If bit13 of the instruction is zero "count" is specified by the least significant five bits of rs2.*
- *If bit13 of the instruction is one "count" is specified by the least significant five bits of the simm13 field (the most significant eight bits of this field should be set to zeroes).*
- *SLL and SRL replace vacated positions with zeroes.*
- *SRA fills vacated positions with the most significant bit of rs1.*

Assembly Representation

sethi Instruction

Makes zeroes the least significant bits of **rd** and place **imm22** in the most significant bits of rd.

When the **rd** and **imm22** fields are zeroes it acts like a **NOP** instruction.

Assembly representation:

sethi 8234, %r24

sethi 0b11110000111100000111100, %r12

SPARC V8 Architecture

Load/Store Instructions

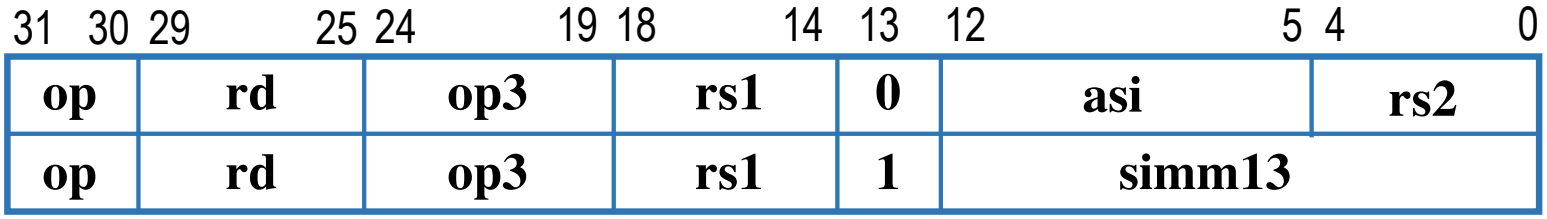
Load/Store Instructions

- The only instructions that addressed memory
- Three types:
  - Load/Store Integer
    - Normal Space
    - Alternate Address Space (implementation dependent)
  - Load/Store Floating Point
  - Load/Store Coprocessor

Dealing with Load/Store Instructions

- Operation
- Calculation of Effective Address
- Size of data (byte, halfword, word, doubleword)
- Signed or unsigned transfer

Load/Store Format



- op = 11
- Load Operation:

  rd <= M[rs1 + rs2, or rs1 + simm13 (sign extended)]
- Store Operation

  M[rs1 + rs2, or rs1 + simm13 (sign extended)] <= rd

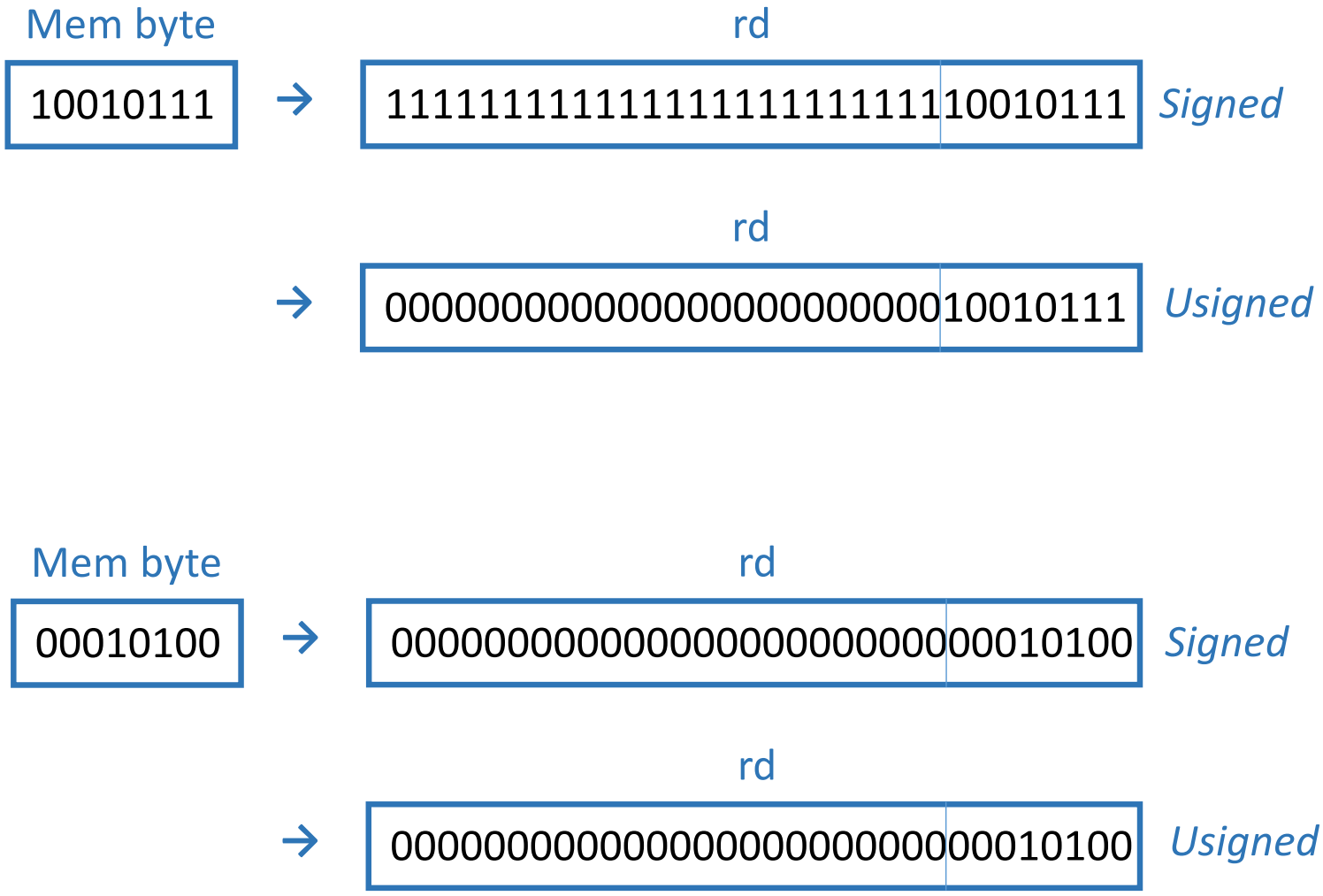
Load/Store Integer Instructions (Normal Space)

Instruction	op3	Operation
lsb	001001	Load sign byte
ldsh	001010	Load sign halfword
ld	000000	Load word
ldub	000001	Load unsigned byte
lduh	000010	Load unsigned halfword
ldd	000011	Load double
stb	000101	Store byte
sth	000110	Store halfword
st	000100	Store word
std	000111	Store double
ldstub	001101	Atomic Load-Store Unsigned Byte
swap	001111	Swap register with memory

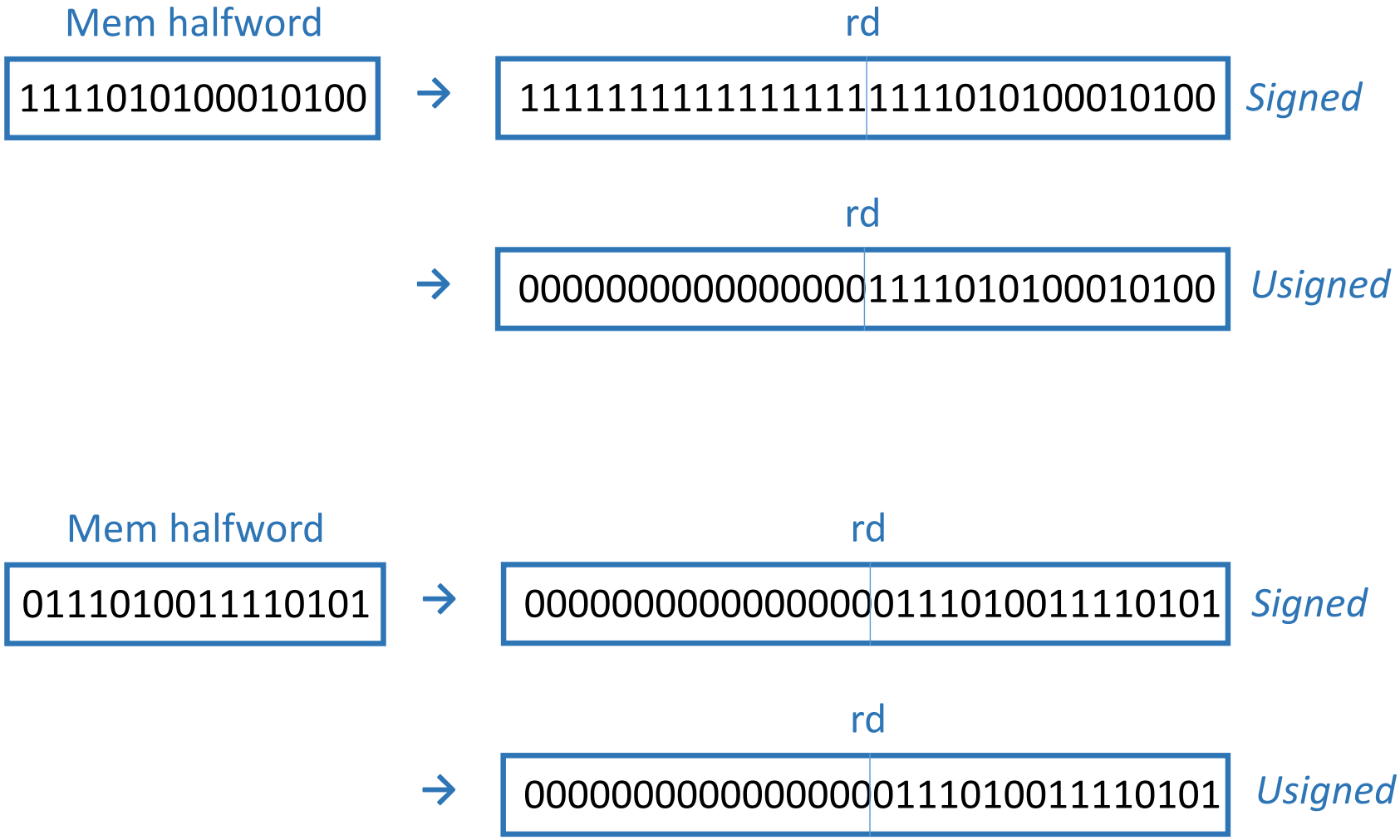
Signed/Unsigned Transfer

- Applies only to load byte and load halfword instructions
- Signed transfer – most significant bits of **rd** are filled with the value of the most significant bit of the byte or halfword.
- Unsigned transfer – most significant bits of **rd** are filled with zeroes.

Signed/Unsigned Byte Transfer



Signed/Unsigned Halfword Transfer



Assembly Representation

SPARC V8 Architecture

Branch Instructions

Branch Instructions

- Are delayed, if the condition is met, transfer to the target address takes place after the instruction following the branch is executed.
- Types
  - Integer
  - Floating point
  - Coprocessor

Branch on Integer Condition Codes Format

Conditional branches

condition is true:

PC <= nPC,

nPC <= PC + 4*disp22 } *nonblocking*

condition is false:

a=0: PC <= nPC,

nPC <= nPC + 4

a=1: PC <= nPC + 4

nPC <= nPC + 8 *Instruction following the branch is annuled*

Unconditional branches (Branch Always and Branch Never) with a=1:

PC <= nPC + 4 *Instruction following the branch is annuled*

nPC <= nPC + 8

Branch on Integer Condition Codes Instructions

Instruction	cond	Operation	Condition
ba	1000	Banch always	
bn	0000	Branch never	
bne	1001	Branch on not equal	Not Z
be	0001	Branch on equal	Z
bg	1010	Branch on greater	Not(Z OR(N XOR V))
ble	0010	Branch on less or equal	Z OR (N XOR V)
bge	1011	Branch on greater or equal	Not (N XOR V)
bl	0011	Branch on less	N xor V
bgu	1100	Branch on greater unsigned	Not (C OR Z)
bleu	0100	Branch on less or equal unsigned	C OR Z
bcc	1101	Branch on Carry =0	Not C
bcs	0101	Branch on Carry =1	C
bpos	1110	Branch on positive	Not N
bneg	0110	Branch on negative	N
bvc	1111	Branch overreflow =0	Not V
bvs	0111	Branch overreflow =1	V

Assembly Representation

label

bne THERE

label

bvs,a LEJOS

Delayed Control-Transfer Couples (DCTI)

- A conditional branch instruction can not follow a conditional branch instruction
- An unconditional branch instruction may follow a conditional branch instruction
- The execution sequence depends on the combination of control transfer instructions (CTI)

DCTI Order of Execution

Case	12 CTI 40	16 CTI 60	Order of Execution
1	DCTI Uncond.	DCTI taken	12,16,40,60,64, …
2	DCTI Uncond.	B*cc(a=0) untaken	12,16,40,44, …
3	DCTI Uncond.	B*cc(a=1) untaken	12,16,44,48, … (40 annulled)
4	DCTI Uncond.	B*A(a=1)	12,16,60,64, … (40 annulled)
5	B*A(a=1)	any CTI	12,40,44, … (16 annulled)
6	B*cc	DCTI	12, unpredictable

B*A	BA, FBA or CBA
B*cc	Bicc, FBfcc, or CBccc
DCTI uncond.	CALL, JMPL, RETT, or B*A (con a=0)
DCTI taken	CALL, JMPL, RETT, B*A (con a=0), or B*cc taken

SPARC V8 Architecture

Subroutine Instructions

Instructions for jumping to a subroutine

call

- A relative jump to subroutine (jumps relative to the PC)
- The number of forward/backward instructions where the jump takes place is determined by an immediate displacement of 30 bits.
- Very convenient for relocating code (no need for recalculating the target address when the code is moved to another memory area)

jmpl

- A direct jump to subroutine
- Normally used for returning from subroutines and traps
- No good for relocation if use for jumping to subroutines (the effective address needs to be recalculated)

call instruction

Operation:

r15 <= PC

PC <= nPC

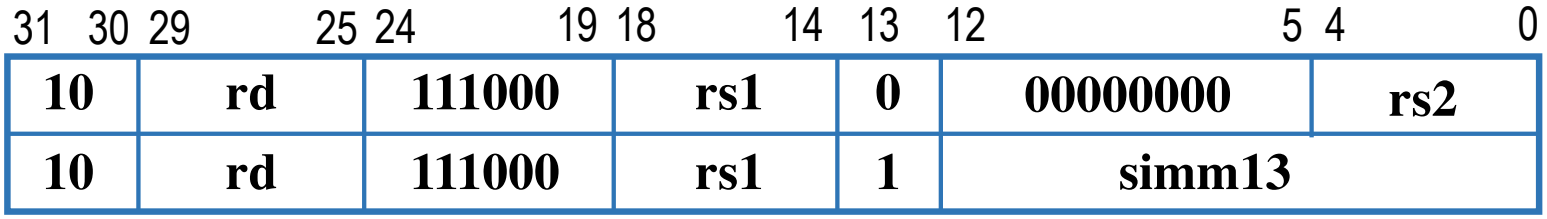
nPC <= PC + 4*disp30

nonblocking

Assembly representation:

call FORWARD ← label

jmpl instruction



31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	111000	rs1	0	00000000	rs2
10	rd	111000	rs1	1	simm13	

Operation:

rd <= **PC**

PC <= **nPC**

nPC <= rs1 + rs2 or rs1 + simm13

nonblocking

Assembly representation:

jmpl FAR, %r8 ← rd

jmpl %r4, %r6, %r1

jmpl %r15 + 4, %r0 = ret (*Return from subroutine*)

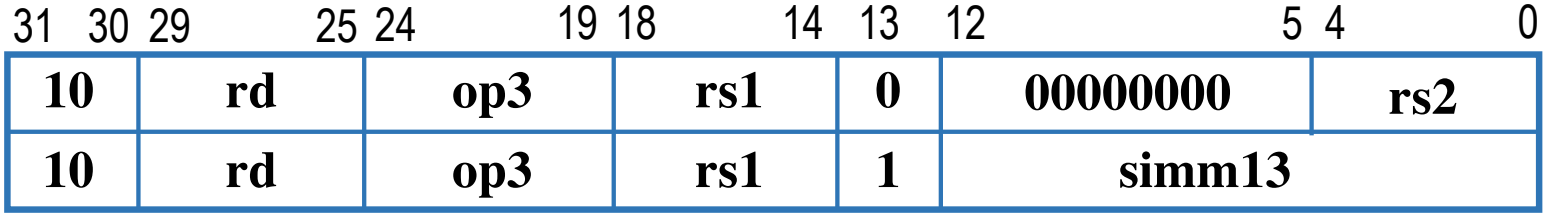
save and restore Instructions

- Control the Register Window Mechanism

save: If WIM(CWP-1) = 0 then
CWP <= CWP-1;
else initiate Window Overflow Trap

restore: If WIM(CWP+1) = 0 then
CWP <= CWP+1;
else initiate Window Underflow Trap

save and restore Instruction Format



save: op3 = 111100

restore: op3 = 111101

If an overflow or underflow trap is not generated, **save** and **restore** behave like normal ADD instructions. However, the source operands are read from the original window while the result is written into **rd** on the new window.

Assembly Representation

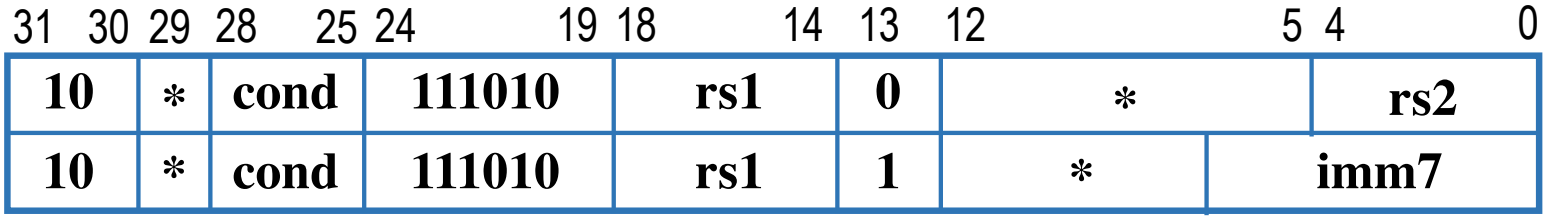
```
save  %r4, %r5, %r1

restore  %r4, -30, %r28
```

SPARC V8 Architecture

Trap Instructions

Trap on Integer Condition Codes Format

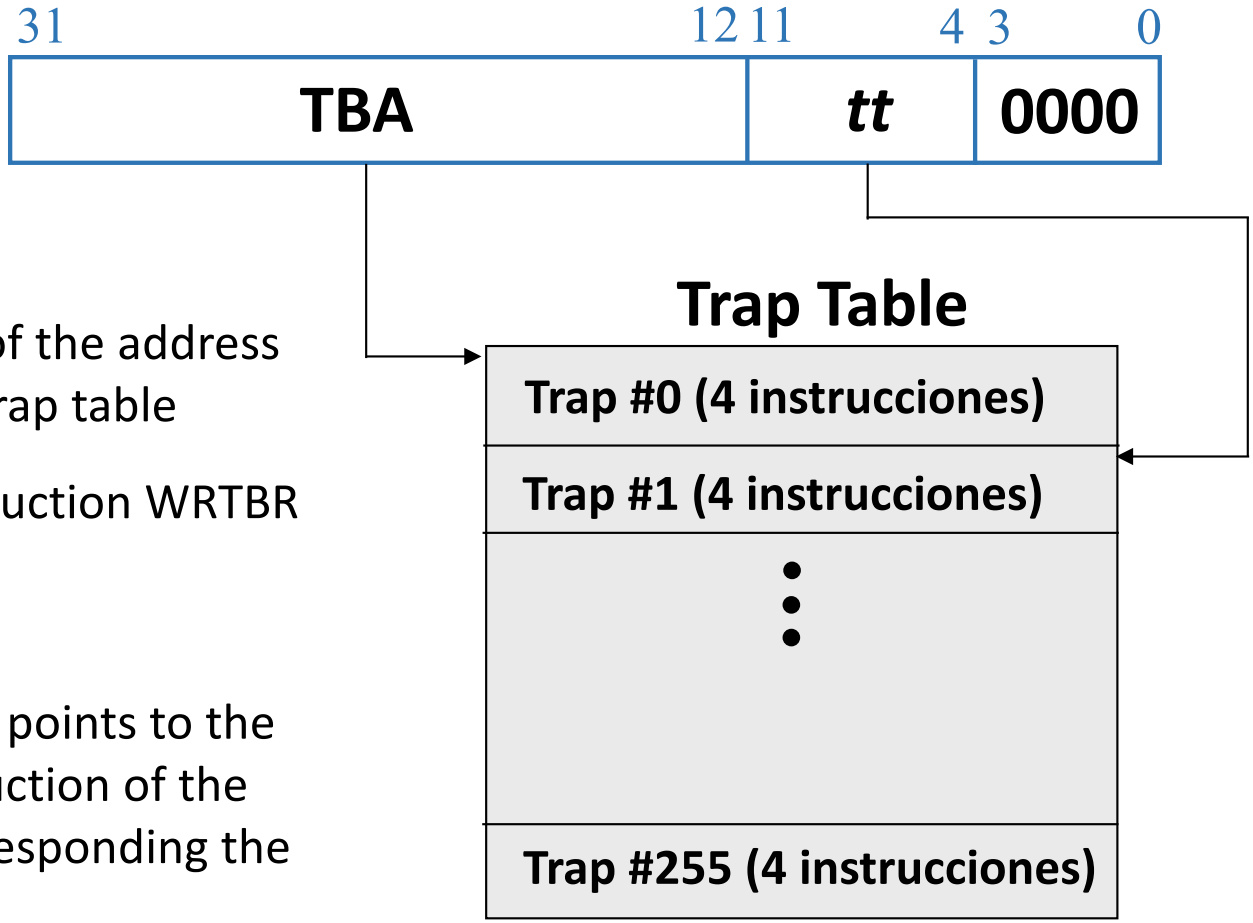


* *reserved*

If the condition is true and there are no pending exceptions or interrupts with higher priority a trap takes place. Otherwise it behaves like a NOP.

If the trap is generated the ***tt*** field of the TBR is written with 128 plus the seven least significant bits of rs1 + rs2 or rs1 + imm7.

Trap Base Register



TBA - Trap Base Address

- Most-significant 20 bits of the address of the beginning of the trap table
- Written by privilege instruction WRTBR

tt - Trap identifier

- Offset from the TBA that points to the address of the first instruction of the trap serving routine corresponding the trap
- Written by the hardware when a trap occurs

General Operation of Traps

- Write ***tt*** field of **TBR**
- Disable traps (**ET** <= 0)
- Preserve processor mode (**PS** <= **S**)
- Change processor mode to supervisor (**S** <= 1)
- Advance to new window (**CWP** <= **CWP** – 1)
- Save **PC** and **nPC** (r17 <= **PC**, r18 <= **NPC**)
- If trap is reset trap **PC** <= 0, **nPC** <= 4
- else **PC** <= **TBR**, **nPC** <= **TBR** + 4

Trap on Integer Condition Codes Instructions

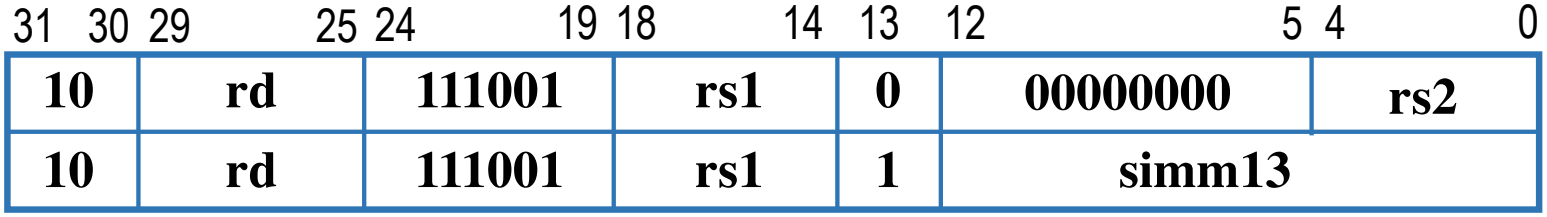
Instruction	cond	Operation	Condition
ta	1000	Trap always	
tn	0000	Trap never	
tne	1001	Trap on not equal	Not Z
te	0001	Trap on equal	Z
tg	1010	Trap on greater	Not(Z OR(N XOR V))
tle	0010	Trap on less or equal	Z OR (N XOR V)
tge	1011	Trap on greater or equal	Not (N XOR V)
tl	0011	Trap on less	N xor V
tgu	1100	Trap on greater unsigned	Not (C OR Z)
tleu	0100	Trap on less or equal unsigned	C OR Z
tcc	1101	Trap on Carry =0	Not C
tcs	0101	Trap on Carry =1	C
tpos	1110	Trap on positive	Not N
tneg	0110	Trap on negative	N
tvc	1111	Trap overreflow =0	Not V
tvs	0111	Trap overreflow =1	V

Assembly Representation

tvc 136

ta 200

Return from trap instruction - rett



31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	111001	rs1	0	00000000	rs2
10	rd	111001	rs1	1	simm13	

Operation:

if WIM(CWP-1)=0

CWP <= CWP+1;

PC <= nPC

nPC = rs1+rs2 or rs1+simm13 (*sign extended*)

S <= PS //restore processor mode

ET <= 1 //enable traps

else trap

**Privileged instruction*

Return from trap instruction - rett

rett must be preceded by jmpl instruction

To reexecute the trapped instruction when returning from a trap handler use the sequence:

```
jmpl %r17,%r0  ! oldPC
rett %r18  ! oldnPC
```

To return to the instruction after the trapped instruction use the sequence:

```
jmpl %r18,%r0  ! oldnPC
rett %r18+4   ! old nPC + 4
```

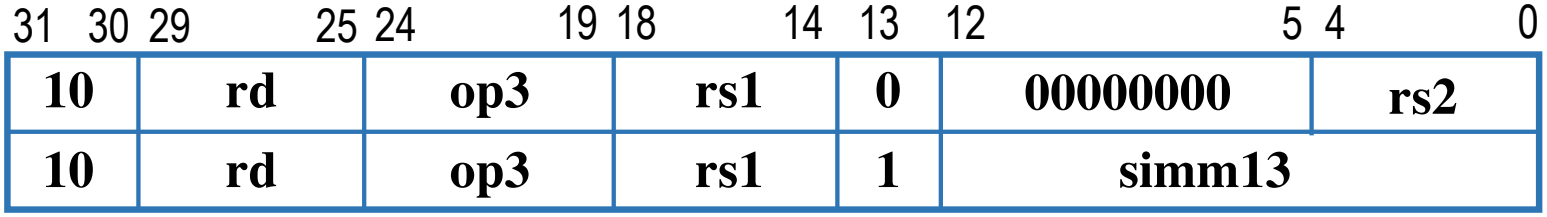
SPARC V8 Architecture

Read/Write State Register Instructions

Read State Register Instructions

Instruction	op3	Operation	Assembly Representation
RDPSR	101001	rd <= PSR	rd %psr, %r23
RDWIM	101010	rd <= WIM	rd %wim, %r19
RDTBR	101011	rd <= TBR	rd %tbr, %r3

Write State Register Instructions



Instruction	op3	Operation	Assembly Representation
WDPSR	110001	PSR<= rs1 **xor** rs2, or PSR<= rs1 **xor** simm13	wr %r5, %r17, %psr
WDWIM	110010	WIM<= rs1 **xor** rs2, or WIM<= rs1 **xor** simm13	wr %r16, 55, %wim
WDTBR	110011	TBR<= rs1 **xor** rs2, or TBR<= rs1 **xor** simm13	wr %r4, %r30, %tbr

*Write state register instructions are **delayed-write** instructions. Depending on implementation writing to the register may take up to three instructions.*

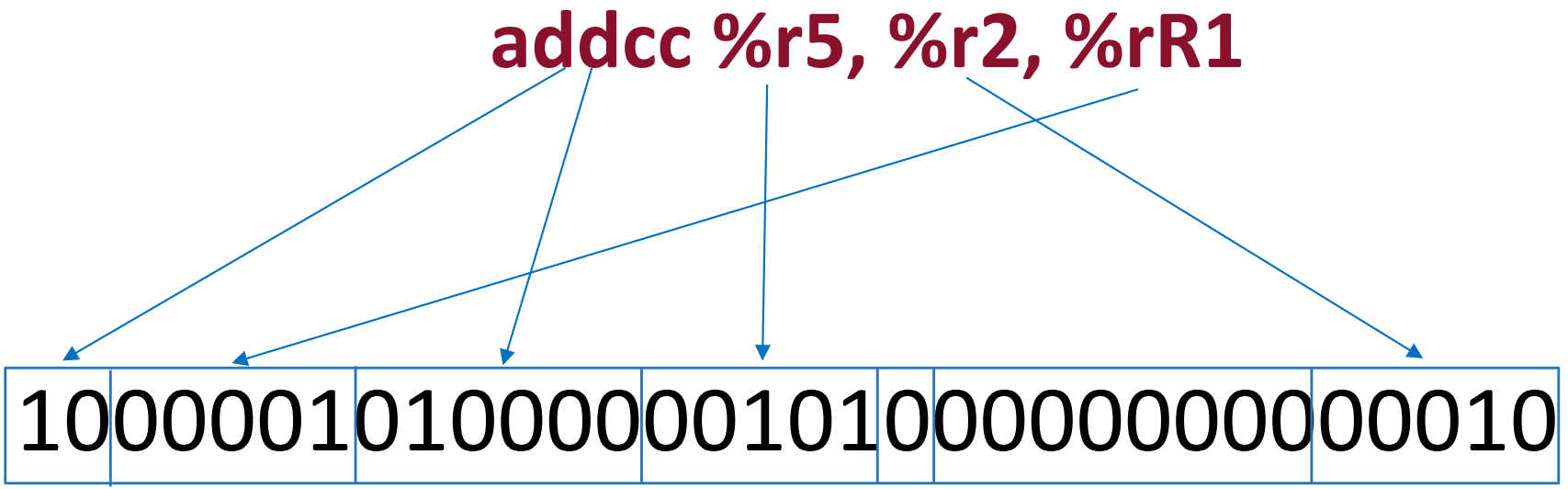
SPARC V8 Architecture

Coding and Decoding Instructions

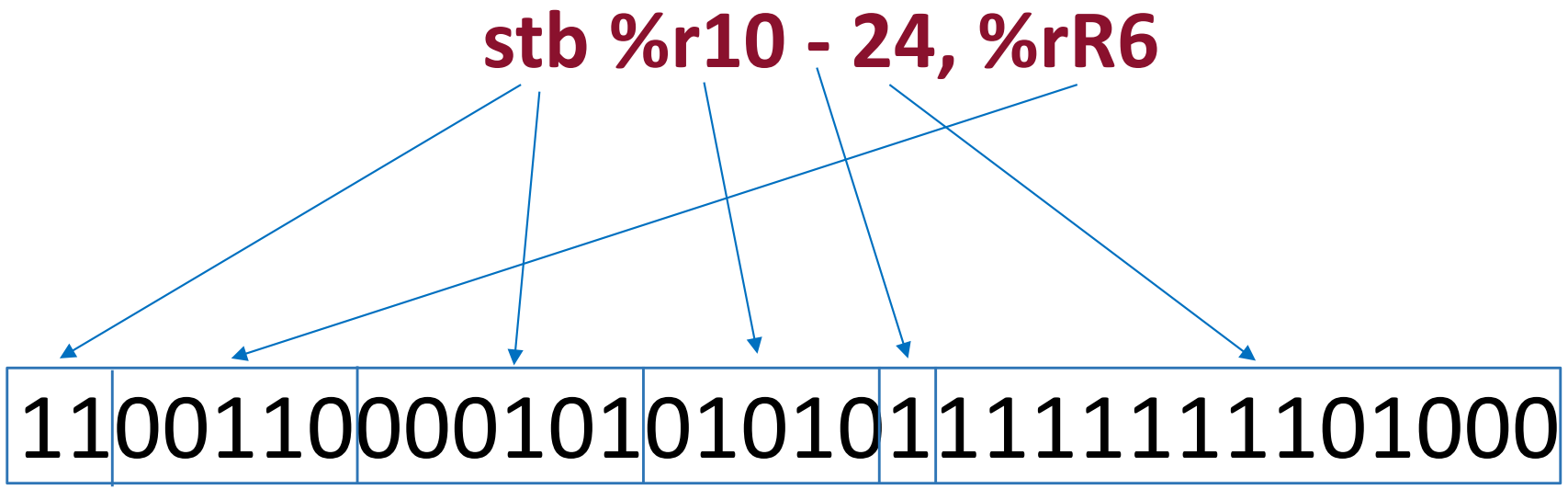
Instruction Coding (Machine Code)

- It's a game of mapping operations on information to a binary number system
- The operations and the information are represented by binary codes

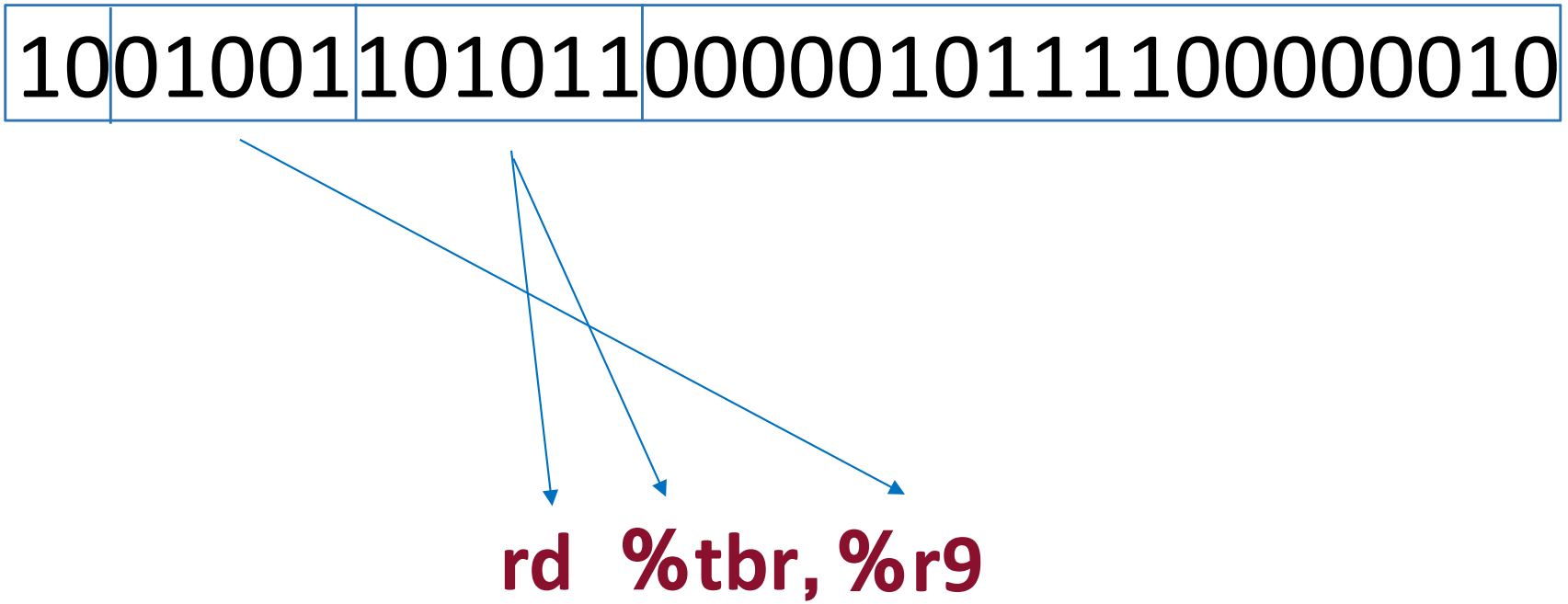
Instruction Coding: Example 1



Instruction Coding: Example 2



Examples of Coding in ARM



Examples of Coding in ARM