Chapter 3: Some Real Machines

Topics

Practical Aspects of Machine Cost-Effectiveness

- Cost for useful work is fundamental issue
- Mounting, case, keyboard, etc. are dominating the cost of integrated circuits
- Upward compatibility preserves software investment
  - Binary compatibility
  - Source compatibility
  - Emulation compatibility
- Performance: strong function of application

Performance Measures

- **MIPS: Millions of Instructions Per Second**
  - **Same job may take more instructions on one machine than on another**
- **MFLOPS: Million Floating Point OPs Per Second**
  - **Other instructions counted as overhead for the floating point**
- **Whetstones: Synthetic benchmark**
  - **A program made up to test specific performance features**
- **Dhrystones: Synthetic competitor for Whetstone**
  - **Made up to "correct" Whetstone's emphasis on floating point**
- **SPEC: Selection of "real" programs**
  - **Taken from the C/Unix world**

CISC Versus RISC Designs

- **CISC: Complex Instruction Set Computer**
  - **Many complex instructions and addressing modes**
  - **Some instructions take many steps to execute**
  - **Not always easy to find best instruction for a task**
- **RISC: Reduced Instruction Set Computer**
  - **Few, simple instructions, addressing modes**
  - **Usually one word per instruction**
  - **May take several instructions to accomplish what CISC can do in one**
  - **Complex address calculations may take several instructions**
  - **Usually has load-store, general register ISA**

Design Characteristics of RISCs

- Simple instructions can be done in few clocks
  - Simplicity may even allow a shorter clock period
- A pipelined design can allow an instruction to complete in every clock period
- Fixed length instructions simplify fetch and decode
- The rules may allow starting next instruction without necessary results of the previous
  - Unconditionally executing the instruction after a branch
  - Starting next instruction before register load is complete

Other RISC Characteristics

- **Prefetching of instructions. (Similar to I8086.)**
- **Pipelining: beginning execution of an instruction before the previous instruction(s) have completed. (Will cover in detail in Chapter 5.)**
- **Superscalar operation—issuing more than one instruction simultaneously. (Instruction-level parallelism. Also covered in Chapter 5.)**
- **Delayed loads, stores, and branches. Operands may not be available when an instruction attempts to access them.**
- **Register windows—ability to switch to a different set of CPU registers with a single command. Alleviates procedure call/return overhead. Discussed with SPARC in this chapter.**

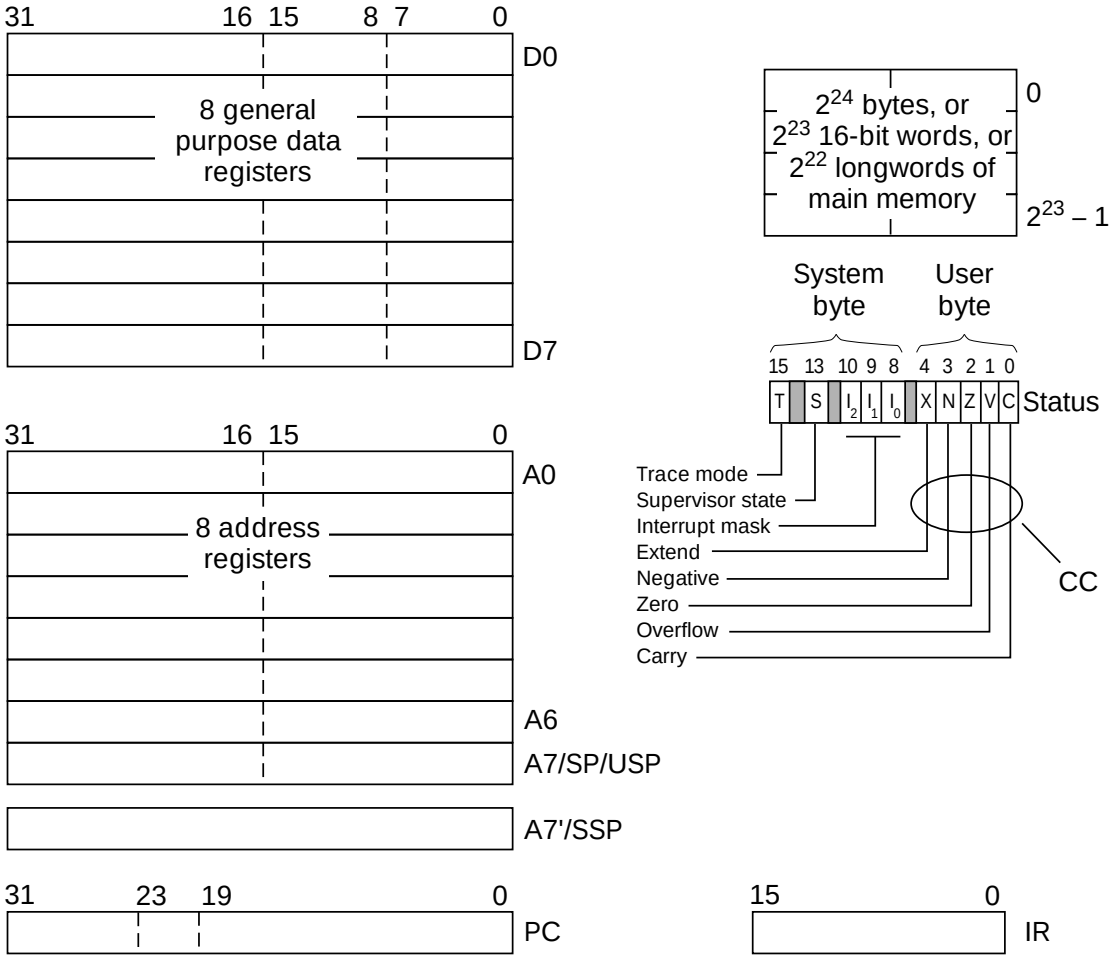
Tbl 3.1 Order of Presenting or Developing a Computer ISA

- ***Memories: structure of data storage in the computer***
  - **Processor-state registers**
  - **Main memory organization**
- ***Formats and their interpretation: meanings of register fields***
  - **Data types**
  - **Instruction format**
  - **Instruction address interpretation**
- ***Instruction interpretation: things done for all instructions***
  - **The fetch-execute cycle**
  - **Exception handling (sometimes deferred)**
- ***Instruction execution: behavior of individual instructions***
  - **Grouping of instructions into classes**
  - **Actions performed by individual instructions**

CISC: The Motorola MC68000

- Introduced in 1979
- One of first 32-bit microprocessors
  - Means that most operations are on 32-bit internal data
  - Some operations may use different number of bits
  - External data paths may not all be 32 bits wide
    - MC68000 had a 24-bit address bus
- Complex Instruction Set Computer—CISC
  - Large instruction set
  - 14 addressing modes

Fig 3.1 The MC68000 Processor State



Features of the 68000 Processor State

- Distinction between 32-bit data registers and 32-bit address registers
- 16-bit instruction register
  - Variable length instructions handled 16 bits at a time
- Stack pointer registers
  - User stack pointer is one of the address registers
  - System stack pointer is a separate single register
    - Discuss: Why a separate system stack
- Condition code register: System and user bytes
  - Arithmetic status (N, Z, V, C, X) is in user status byte
  - System status has supervisor and trace mode flags, as well as the interrupt mask

RTN Processor State for the MC68000

D[0..7]⟨31..0⟩:	General purpose data registers
A[0..7]⟨31..0⟩:	Address registers
A7´⟨31..0⟩:	System stack pointer
PC⟨31..0⟩:	Program counter
IR⟨15..0⟩:	Instruction register
Status⟨15..0⟩:	System status byte and user status byte
SP := A[7]:	User stack pointer, also called USP
SSP := A7´:	System stack pointer
C := Status⟨0⟩: V := Status⟨1⟩:	Carry and Overflow flags
Z := Status⟨2⟩: N := Status⟨3⟩:	Zero and Negative flags
X := Status⟨4⟩:	Extend flag
INT⟨2..0⟩ := Status⟨10..8⟩:	Interrupt mask in system status byte
S := Status⟨13⟩: T := Status⟨15⟩:	Supervisor state and Trace mode flags

Main Memory in the MC68000

Main memory:

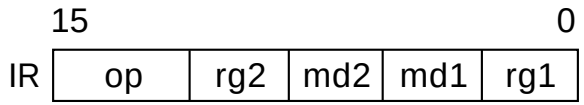
Mb[0..2^{24}-1]⟨7..0⟩:	**Memory as bytes**
Mw[ad]⟨15..0⟩ := Mb[ad]#Mb[ad+1]:	**Memory as words**
Ml[ad]⟨31..0⟩ := Mw[ad]#Mw[ad+2]:	**Memory as long words**

- **The word and longword forms are "big-endian"**
  - **The lowest numbered byte contains the most significant bit (big end) of the word**
- **Words and longwords have "hard" alignment constraints not described in the above RTN**
  - **Word addresses must end in one binary 0**
  - **Longword addresses must end in two binary zeros**

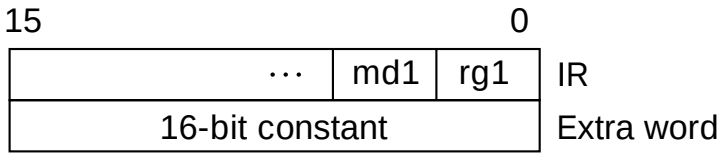
MC68000 Supports Several Operand Types

- Like many CISC machines, the 68000 allows one instruction to operate on several types
  - MOVE.B for bytes, MOVE.W for words, and MOVE.L for longwords; also ADD.B, ADD.W, ADD.L, etc.
  - Operand length is coded as bits of the instruction word
- Bits coding operand type vary with instruction
  - For use with RTN descriptions, we assume a function d := datalen(IR) that returns 1, 2, or 4 for operand length

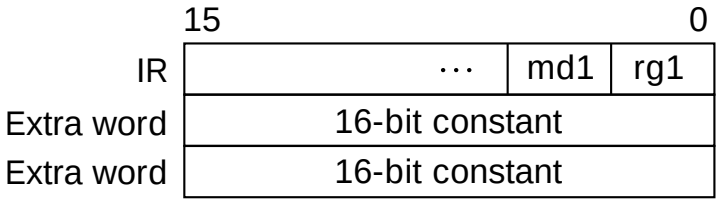
Fig 3.2 Some MC68000 Instruction Formats



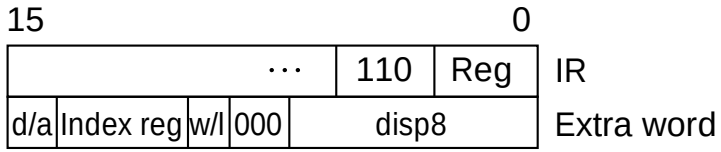
(a) A 1-word move instruction



(b) A 2-word instruction



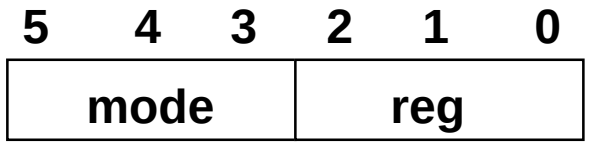
(c) A 3-word instruction



(d) Instruction with indexed address

General Form of Addressing Modes in the MC68000

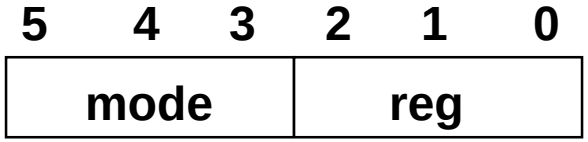
- A general address of an operand or result is specified by a 6-bit field with mode and register numbers



Provides access paths to operands

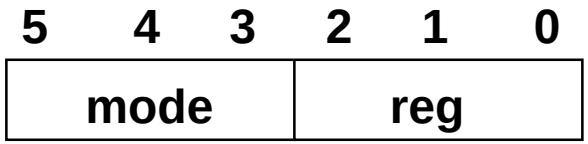
- Not all operands and results can be specified by a general address: some must be in registers
- Not all modes are legal in all parts of an instruction

Tbl 3.2 MC68000 Addressing Modes



Name	Mode	Reg.	Assembler	Extra Words	Brief description
Data reg. direct	0	0-7	Dn	0	Dn
Addr. reg. direct	1	0-7	An	0	An
Addr. reg. indirect	2	0-7	(An)	0	M[An]
Autoincrement	3	0-7	(An)+	0	M[An];An←An+d
Autodecrement	4	0-7	-(An)	0	An←An-d;M[An]
Based	5	0-7	disp16(An)	1	M[An+disp16]
Based indexed short	6	0-7	disp8(An,XnLo)	1	M[An+XnLo+disp8]
Based indexed long	6	0-7	disp8(An,Xn)	1	M[An+Xn+disp8]
Absolute short	7	0	addr16	1	M[addr16]
Absolute long	7	1	addr32	2	M[addr32]
Relative	7	2	disp16(PC)	1	M[PC+disp16]
Rel. indexed short	7	3	disp8(PC,XnLo)	1	M[PC+XnLo+disp8]
Rel. indexed long	7	3	disp8(PC,Xn)	1	M[PC+Xn+disp8]
Immediate	7	4	#data	1-2	data

RTN Description of MC68000 Addressing

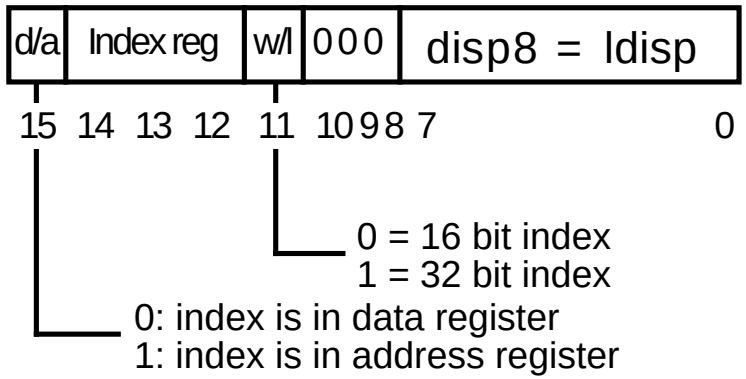


- The addressing modes interpret many items
  - The instruction: in the IR register
  - The following 16-bit word: described as Mw[PC]
  - The D and A registers in the CPU
- Many addressing modes calculate an effective memory address
- Some modes designate a register
- Some modes result in a constant operand
- There are restrictions on the use of some modes

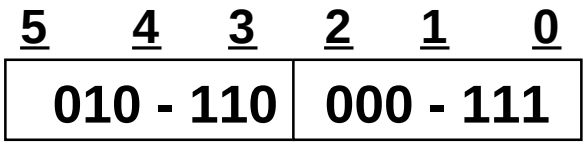
RTN Formatting for Effective Address Calculation

```
XR[0..15]⟨31..0⟩ :=
        D[0..7]⟨31..0⟩ # A[0..7]⟨31..0⟩:      Index register can be D or A;
xr⟨3..0⟩ := Mw[PC]⟨15..12⟩:                   Index specifier for index mode;
wl := Mw[PC]⟨11⟩:                             Short or long index flag;
dsp8⟨7..0⟩ := Mw[PC]⟨7..0⟩:                   Displacement for index mode;
index := (  (wl=0) → XR[xr]⟨15..0⟩:           Short or
            (w1=1) → XR[xr]⟨31..0⟩):                  long index value;
```

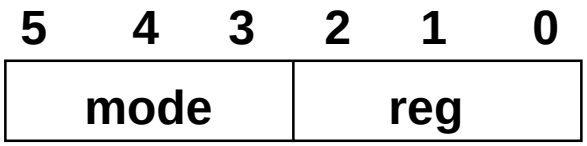
- Either an A or a D register can be used as an index
- A 4-bit field in the 2nd instruction word specifies the index register
- Low order 8-bits of 2nd word are used as offset
- Either 16 or 32 bits of index register may be used



Modes That Calculate a Memory Address Using a Register

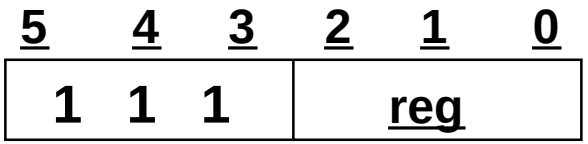


- md and rg are the 3-bit mode and register fields
- ea stands for *effective address*




```
ea(md, rg) := (
  (md = 2) → A[rg⟨2..0⟩]:                                  Mode 2 is
                                                             A register indirect;
  (md = 3) →                                               Mode 3 is
        (A[rg⟨2..0⟩]; A[rg⟨2..0⟩] ←A[rg⟨2..0⟩] + d):          autoincrement;
  (md = 4) →                                               Mode 4 is
        (A[rg⟨2..0⟩] ←A[rg⟨2..0⟩] - d; A[rg⟨2..0⟩]):          autodecrement;
  (md = 5) →                                               Mode 5 is based
        (A[rg⟨2..0⟩] + Mw[PC]; PC ← PC + 2):                 or offset addressing;
  (md = 6) →                                               Mode 6 is based
        (A[rg⟨2..0⟩] + index + dsp8; PC ←PC + 2):          indexed addressing;
```

Mode 7 Uses the Register Field to Expand the Number of Modes

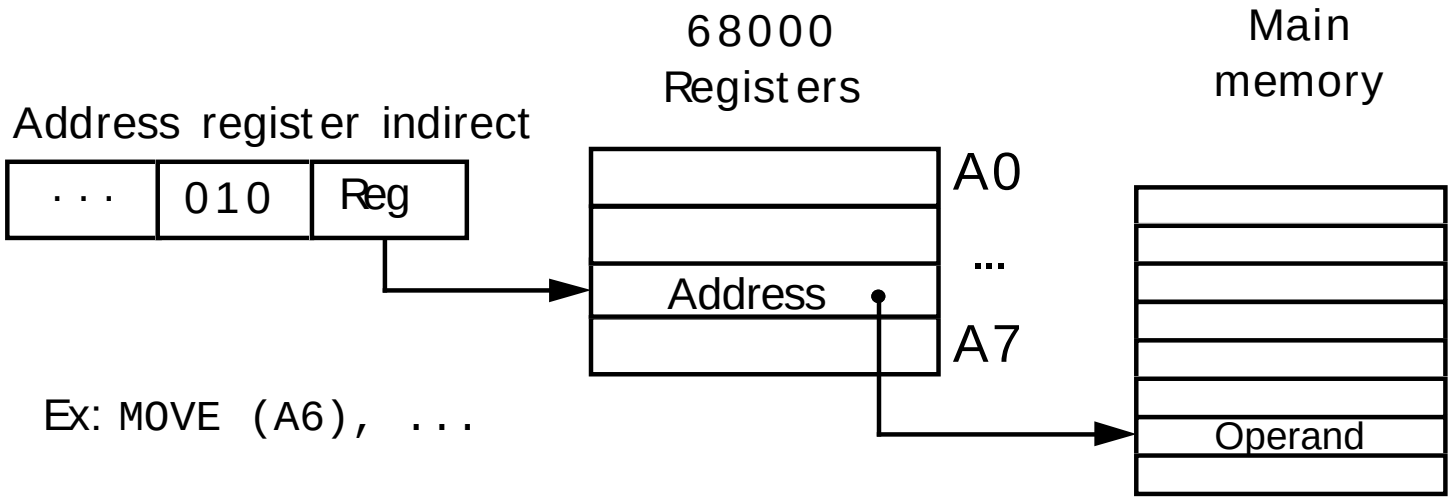


- These modes still calculate a memory address

ea (md, rg) :=
. . .
(md = 7 ∧ rg = 0) → Mode 7, register 0 is
(Mw[PC]{sign extend to 32 bits}; PC ← PC + 2): short absolute;
(md = 7 ∧ rg = 1) → Mode 7, register 1 is
(Ml[PC]; PC ← PC + 4): long absolute;
(md = 7 ∧ rg = 2) → Mode 7, register 2 is
(PC + Mw[PC]{sign extend to 32 bits}; program counter
PC ← PC + 2): relative
addressing;
(md = 7 ∧ rg = 3) → Mode 7, register 3 is
(PC + index + dsp8; PC ← PC + 2)): relative indexed.

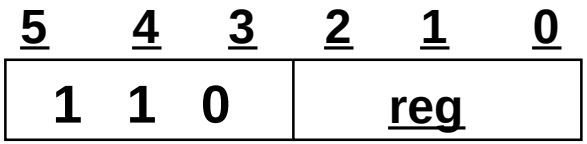
Fig 3.3 Address Register Indirect Addressing

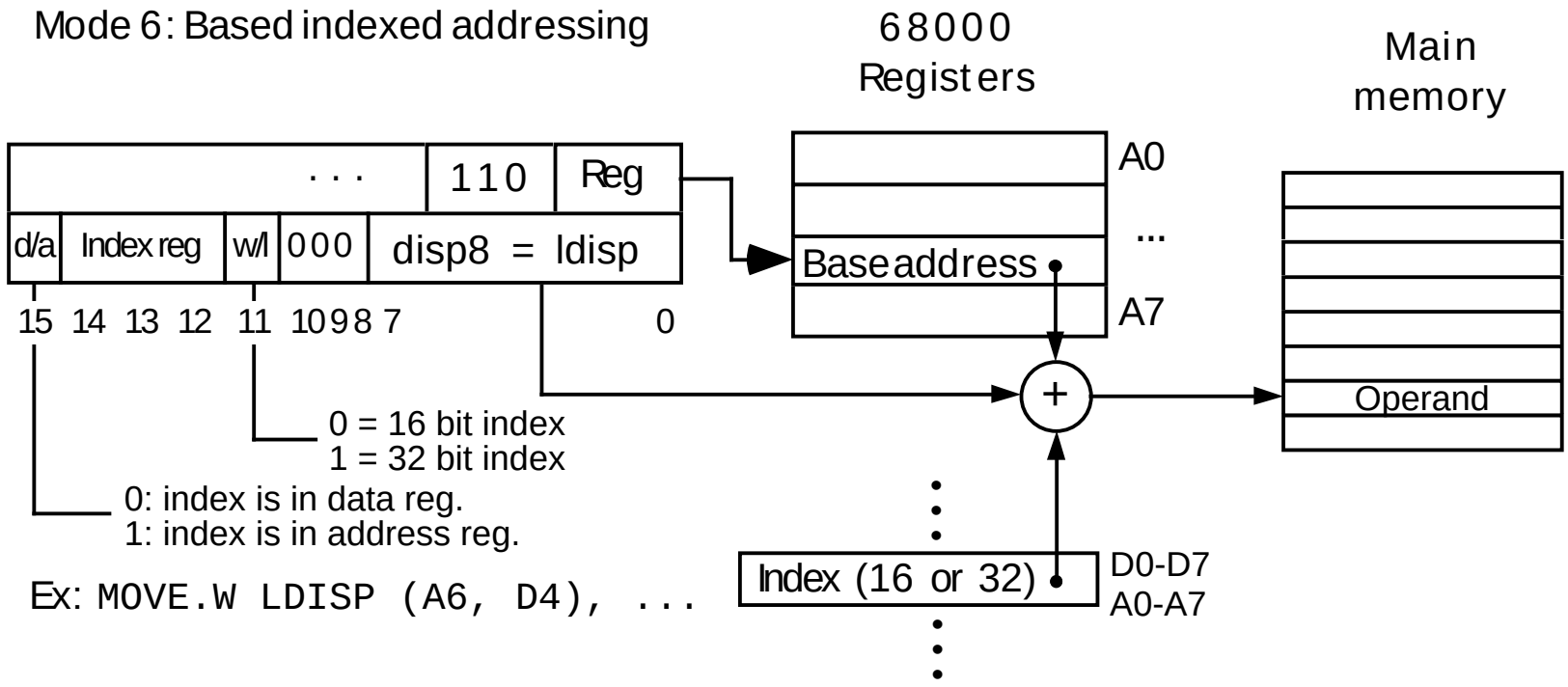
5	4	3	2	1	0
0	1	0	reg		



- **Same picture for autoincrement or decrement**
  - **Address register incremented after address obtained in autoincrement**
  - **Address register decremented before address obtained in autodecrement**

Fig 3.4 Mode 6: Based Indexed Addressing

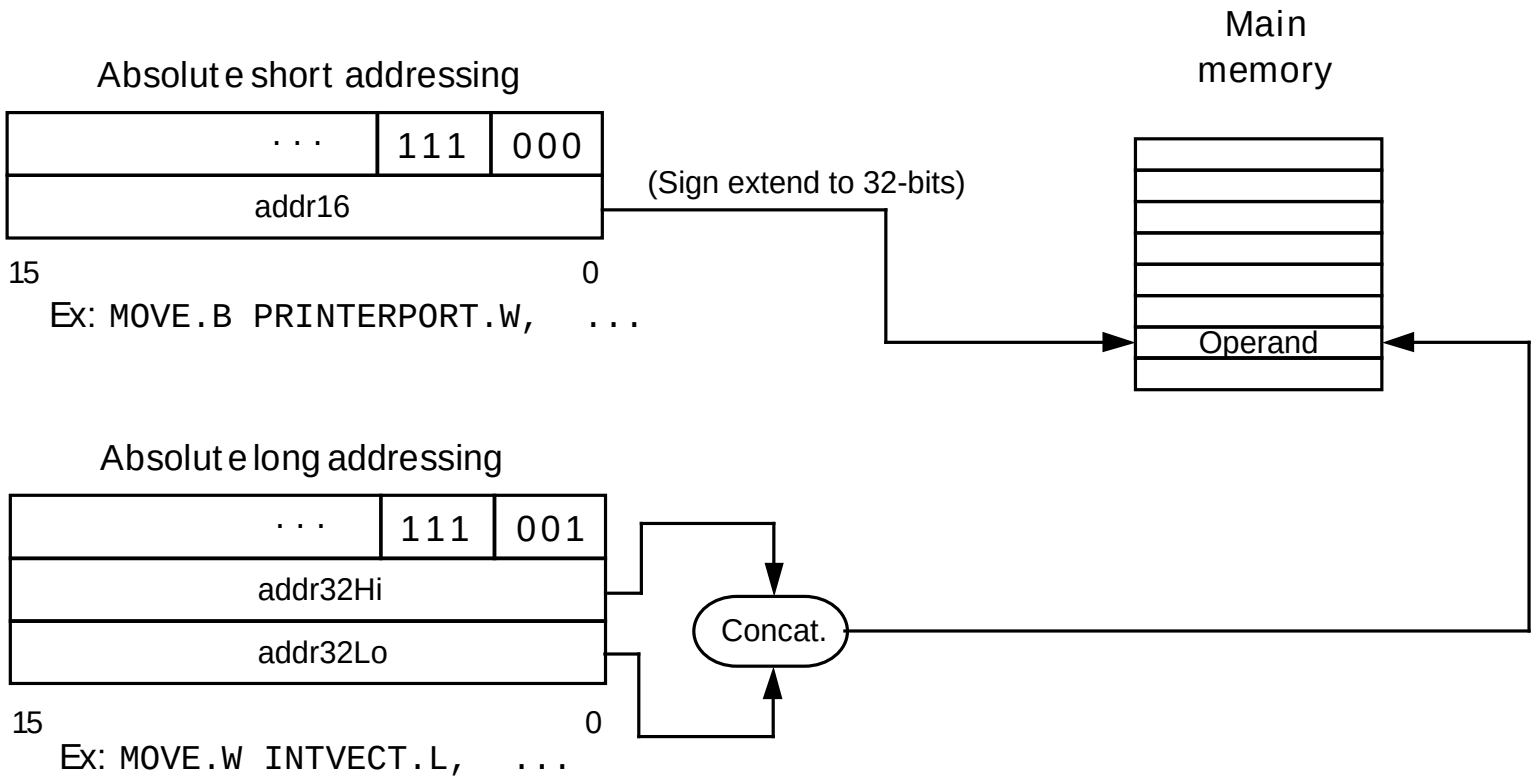




- **Three things are added to get the address**

Mode 7-0,1: Absolute Addressing

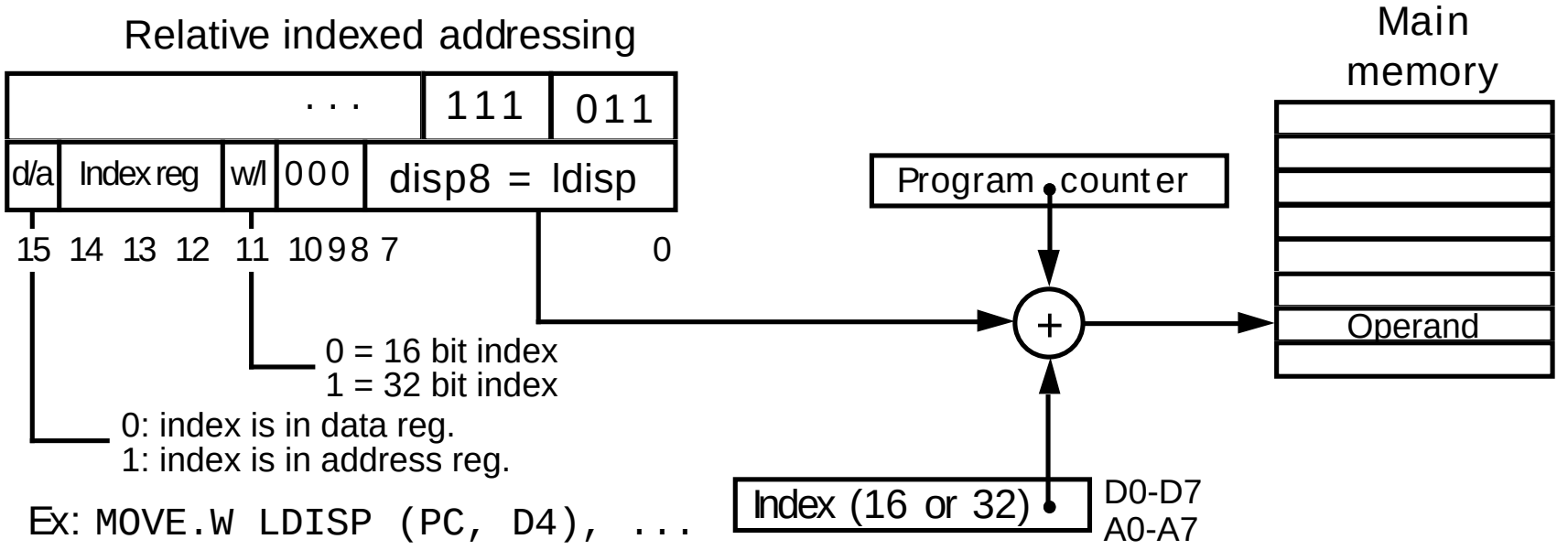
5	4	3	2	1	0
1	1	1	000 (16-bit) 001 (32-bit)		



- **Absolute addresses can be 16 or 32 bits**

Mode 7, Reg 3: Relative Indexed Addressing

5	4	3	2	1	0
1	1	1	0	1	1



- **Same as indexed mode but uses PC instead of A register as base**

Operands in Registers or Memory Can Have Different Lengths

```
memval(md, rg) :=
  ( (md⟨2..1⟩ = 1) ∨ (md⟨2..1⟩ = 2) ∨ (md⟨2..0⟩ = 6) ∨
  ((md⟨2..0⟩ = 7) ∧ (rg⟨2⟩ =0)) ):
opnd(md, rg) := (
  (d=1) → opndb(md, rg): (d=2) → opndw(md, rg):
  (d=4) → opndl(md, rg)                              ):
opndl(md, rg)⟨31..0⟩ := (
                                  . . .     ):
opndw(md, rg)⟨15..0⟩ := (
  memval(md, rg) → Mw[ea(md, rg)]⟨15..0⟩:
  md =0 → D[rg]⟨15..0⟩:
  md = 1 → A[rg]⟨15..0⟩:
  (md = 7 ∧ rg = 4) → (Mw[PC]⟨15..0⟩: PC ← PC+2) ):
opndb(md, rg)⟨7..0⟩ := (
                                  . . .
  (md = 7 ∧ rg = 4) → (Mw[PC]⟨7..0⟩: PC ← PC+2) ):
```

A memory address is used with these modes only.

The operand length in the instruction tells which to use.

A long operand can be

. . .

A word operand is similar but needs only a 16-bit immediate following the instruction word.

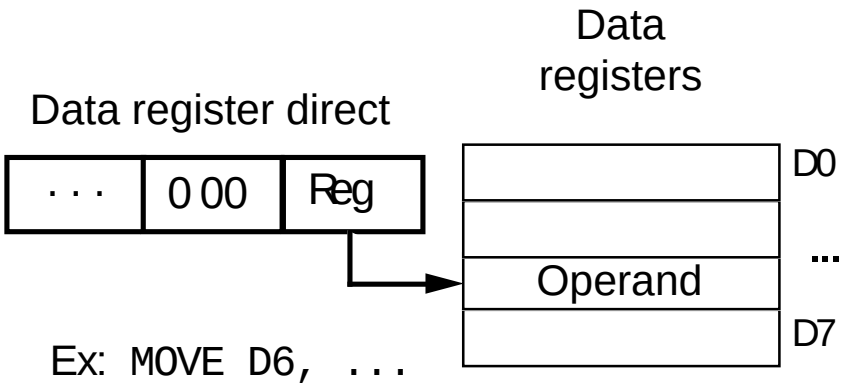
Byte operands

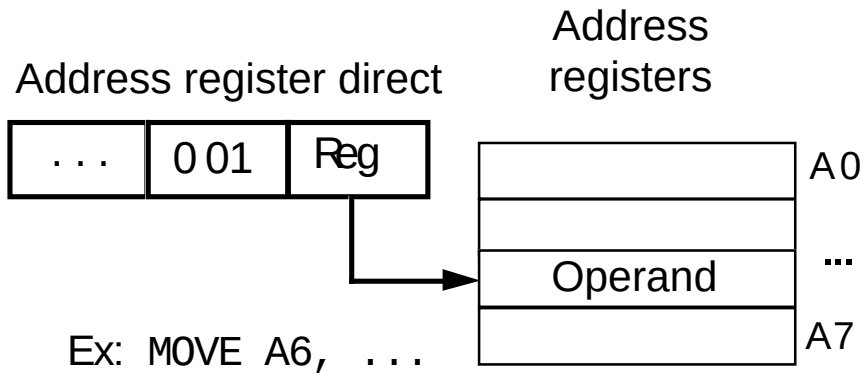
. . .

instruction word.

Modes 0 and 1: Register Direct Addressing

5	4	3	2	1	0
0	0	0 (D)	reg		
0	0	1 (A)			



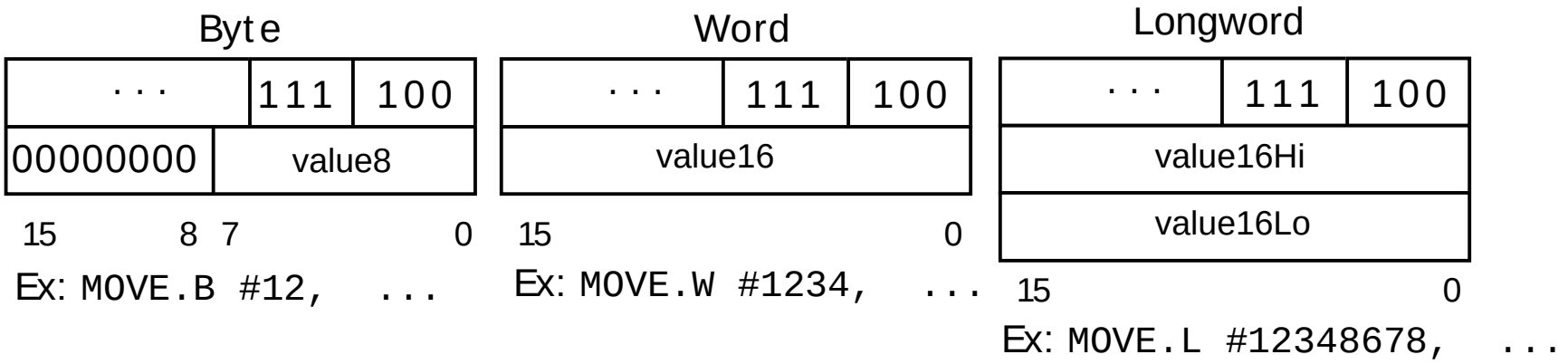


- **The register itself provides a place to store a result or a place to get an operand**
- **There is no memory address with this mode**

Fig 3.5 Mode 7, Reg 4: Immediate Addressing Operands are stored *in the instruction*

5	4	3	2	1	0
1	1	1	1	0	0

Instruction word and 1 or 2 following words



- **Data length is specified by the opcode field, not the Mode/Reg field**

Not Every Addressing Mode Can Be Used for Results

rsltadr(md, rg) := memval(md, rg) $\wedge \neg$(md=7 $\wedge$(rg=2$\vee$rg=3)):

- The MC68000 disallows relative addressing for results
- This is captured in RTN by defining a function that is true (= 1) if the memory address specified by the mode is legal for results
- Register immediate is also legal for results, but will be handled separately

Result Modes Must Have a Place to Write Data: Memory or Register

```
rsltl(md, rg)⟨31..0⟩ := (                                  32-bit result
  rsltadr(md, rg) → Ml[ea(md, rg)]⟨31..0⟩:
  md = 0 → D[rg]⟨31..0⟩:
  md = 1 → A[rg]⟨31..0⟩                    ):
rsltw(md, rg)⟨15..0⟩ := (                                  16-bit result
  rsltadr(md, rg) → Mw[ea(md, rg)]⟨15..0⟩:
  md = 0 → D[rg]⟨15..0⟩:
  md = 1 → A[rg]⟨15..0⟩                    ):
rsltb(md, rg)⟨7..0⟩ := (                                   8-bit result
  rsltadr(md, rg) → Mb[ea(md, rg)]⟨7..0⟩:
  md = 0 → D[rg]⟨7..0⟩:
  md = 1 → A[rg]⟨7..0⟩                     ):
rslt(md, rg) := (                                          The result length in the
  (d=1) → rsltb(md, rg): (d=2) → rsltw(md, rg):                instruction tells
  (d=4) → rsltl(md, rg)                    ):                which to use
```

MC68000 Instruction Interpretation

- Instruction interpretation is simple when exceptions are ignored

Instruction_interpretation := (
Run → ((IR⟨15..0⟩ ← Mw[PC]⟨15..0⟩: PC ← PC + 2);
instruction_execution);):

- **Instructions are fetched 16 bits at a time**
  - **PC is advanced by 2 as each 16-bit word is fetched**
  - **Addressing mode may advance it a total of 2 or 4 or more words, under command from the control unit**

Tbl 3.3 MC68000 Data Movement Instructions

Inst.	Operands →	1st word	XNZVC	Operation	Size
MOVE.B	EAs, EAd	0001dddddddssssss	- x x 0 0	dst ← src	byte
MOVE.W	EAs, EAd	0011dddddddssssss	- x x 0 0	dst ← src	word
MOVE.L	EAs, EAd	0010dddddddssssss	- x x 0 0	dst ← src	long
MOVEA.W	EAs, An	0011rrr001ssssss	- - - - -	An ← src	word
MOVEA.L	EAs, An	0010rrr001ssssss	- - - - -	An ← src	long
LEA.L	EAc, An	0100aaa111ssssss	- - - - -	An ← EA	addr.
EXG	Dx, Dy	1100xxx1mmmmmyyy	- - - - -	Dx ↔ Dy	long

- The op code location and size depends on the instruction (compare to SRC)

RTN for a Typical MC68000 Move Instruction

- **The instruction format for Move includes mode and register for source and destination addresses**

  op⟨3..0⟩ := IR⟨15..12⟩: rg1⟨2..0⟩ := IR⟨2..0⟩: md1⟨2..0⟩ := IR⟨5..3⟩: rg2⟨2..0⟩ := IR⟨11..9⟩: md2⟨2..0⟩ := IR⟨8..6⟩:

  tmp⟨31..0⟩:
  move (:= op⟨3..2⟩ := 0) → (
  tmp ← opnd(md1, rg1);
  (Z ← (tmp=0): N ←(tmp<0): V ← 0: C ← 0):
  rslt(md2, rg2) ← tmp):

  - **The temporary register tmp is used because every invocation of opnd() causes another fetch**

Tbl 3.4 MC68000 Integer Arithmetic and Logic Instructions

Op.	Operands	Inst. word	XNZVC	Operation	Sizes
ADD	EA,Dn	1101rrrmmmaaaaaa	x x x x x	dst ← dst + src	b, w, l
SUB	EA,Dn	1001rrrmmmaaaaaa	x x x x x	dst ← dst - srC	b, w, l
CMP	EA,Dn	1011rrrmmmaaaaaa	- x x x x	dst-src	b, w, l
CMPI	#dat,EA	00001100wwaaaaaa	- x x x x	dst-immed.data	b, w, l
MULS	EA, Dn	1100rrr111aaaaaa	- x x 0 0	Dn←Dn*src	l←w*w
DIVS	EA,Dn	1000rrr111aaaaaa	- x x x 0	Dn←Dn/src	l←l/w
AND	EA,Dn	1100rrrmmmaaaaaa	- x x 0 0	dst←dst∧src	b, w, l
OR	EA,Dn	1000rrrmmmaaaaaa	- x x 0 0	dst←dst∨src	b, w, l
EOR	EA,Dn	1011rrrmmmaaaaaa	- x x 0 0	dst←dst⊕src	b, w, l
CLR	EAs	01000010wwaaaaaa	- 0 1 0 0	dst∧dst	b, w, l
NEG	EAs	01000100wwaaaaaa	- x x x x	dst←0 - dst	b, w, l
TST	EAs	01001010wwaaaaaa	- x x 0 0	dst–0	b, w, l
NOT	EAs	01000110wwaaaaaa	- x x x x	dst← ¬dst	b, w, l

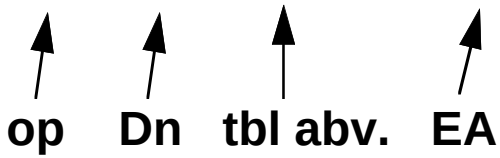
Notes on MC68000 Arithmetic and Logic Instructions

All 2-operand ALU instructions are either D → EA or EA → D. Which is it?

- **Only one operand uses EA**
- **The other operand is always accessed by Data register direct**
- **The 3-bit mmm field specifies whether D is the source or destination, and whether it is B, W, or L**

Byte	Word	Long	Destination
000	001	010	Dn
100	101	110	EA

Ex: SUB EA, Dn: **1011 rrr mmm aaaaaa**



Note: There are several exceptions to the rule above. See text and mfr. data sheet.

RTN Description of a Typical MC68000 Arithmetic Instruction

- **Subtract is a typical arithmetic instruction**
- **Need a temporary register to hold an address**

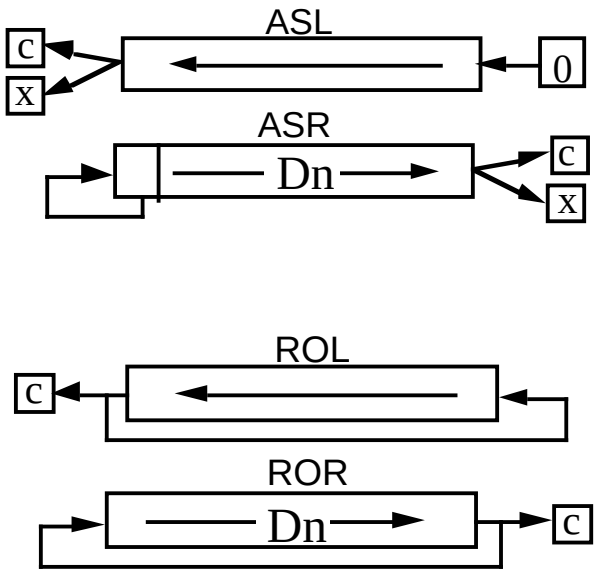
tmp⟨31..0⟩: temporary register for address

```
sub (:= op=9) → (
        (md2⟨2⟩ =0) → D[rg2] ← D[rg2] - opnd(md1, rg1):
        (md2⟨2⟩ =1) → (memval(md1, rg1) → (tmp ← ea(md1, rg1);
                                           M[tmp] ← M[tmp] - D[rg2]  ):
            ¬memval(md1, rg1) → rslt(md1, rg1) ← rslt(md1, rg1) - D[rg2])
                                                                    ):
```

- **This definition does not handle the condition codes**

MC68000 Arithmetic Shifts and Single Word Rotates

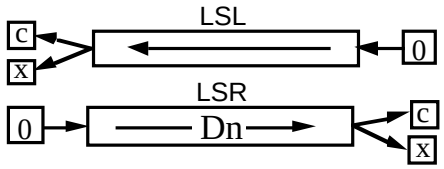
Op.	Operands	Inst. word	XV
ASd	EA	1110000d11aaaaaa	x x
ASd	#cnt,Dn	1110cccdww000rrr	x x
ASd	Dm,Dn	1110RRRdww100rrr	x x
ROd	EA	1110011d11aaaaaa	- 0
ROd	#cnt,Dn	1110cccdww011rrr	- 0
ROd	Dm,Dn	1110RRRdww111rrr	- 0



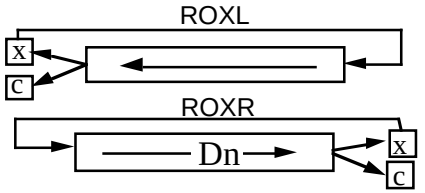
- d is L or R for left or right shift, respectively
- EA form has shift count of 1

MC68000 Logical Shifts and Extended Rotates

Op.	Operands	Inst. word	XV
LSd	EA	1110001d11aaaaaa	x 0
LSd	#cnt,Dn	1110cccdww001rrr	x 0
LSd	Dm,Dn	1110RRRdww101rrr	x 0



ROXd	EA	1110010d11aaaaaa	x 0
ROXd	#cnt,Dn	1110cccdww010rrr	x 0
ROXd	Dm,Dn	1110RRRdww110rrr	x 0



- Field ww specifies byte, word, or longword
- N and Z set according to result, C = last bit shifted out

MC68000 Conditional Branch and Test Instructions

Op.	Operands	Inst. word	Operation
Bcc	disp	0110ccccdddddddd DDDDDDDDDDDDDDDD	if (cond) then PC ← PC + disp
DBcc	Dn,disp	0101cccc11001rrr	if ¬(cond) then Dn←Dn-1 if (Dn≠-1) then PC←PC+disp) else PC ← PC + 2
Scc	EA	0101cccc11aaaaaa	if (cond) then (EA) ←FFH else (EA) ← 00H

- DBcc is used for counted loops with an optional end condition
- Scc sets a byte to the outcome of a test

Conditions That Can Be Evaluated for Branch, Etc.

Code	Meaning	Name	Flag expression
0000	true	T	1
0001	false	F	$\overline{0}$
0100	carry clear	CC	$\overline{C}$
0101	carry set	CS	C
0111	equal	EQ	Z
0110	not equal	NE	$\overline{Z}$
1011	minus	MI	N
1010	plus	PL	$\overline{N}$
0011	low or same	LS	$C+\overline{Z}$
1101	less than	LT	$N\cdot\overline{V}+\overline{N}\cdot V$
1100	greater or equal	GE	$\overline{N}\cdot\overline{V}+N\cdot V$
1110	greater than	GT	$N\cdot\overline{V}\cdot\overline{Z}+\overline{N}\cdot V\cdot\overline{Z}$
1111	less or equal	LE	$\overline{N}\cdot\overline{V}+N\cdot V+Z$
0010	high	HI	$\overline{C}\cdot Z$
1000	overflow clear	VC	$\overline{V}$
1001	overflow set	VS	V

Conditional Branches First Set Condition Codes, Then Branch

```
if ( X = 0 ) goto LOC
```

```
        TST    X       ;ands X with itself and sets N and Z
        BEQ    LOC     ;branch to LOC if X = 0
        .
        .
        .
LOC:
```

- **EQ tests the right condition codes for = 0, as above, or A = B following a compare, CMP A, B**

MC68000 Unconditional Control Transfers

Op.	Operands	Inst. word	Operation
BRA	disp	01100000dddddddd DDDDDDDDDDDDDDDD	PC ← PC + disp
BSR	disp	01100001dddddddd DDDDDDDDDDDDDDDD	-(SP) ← PC; PC ← PC + disp
JMP	EA	0100111011aaaaaa	PC ← EA
JSR	EA	0100111010aaaaaa	-(SP) ← PC; PC ← EA

- Subroutine links push the return address onto the stack pointed to by A7 = SP

MC68000 Subroutine Return Instructions

Op.	Operands	Inst. word	Operation
RTR		0100111001110111	CC ← (SP)+; PC ← (SP)+
RTS		0100111001110101	PC ← (SP)+
LINK	An,disp	0100111001010rrr DDDDDDDDDDDDDDDD	-(SP) ← An; An ← SP; SP ← SP + disp
UNLK	An	0100111001011rrr	SP ← An; An ← (SP)+

- Subroutine linkage uses stack for return address
- LINK and UNLK allocate and de-allocate multiple word stack frames

MC68000 Assembly Code Example: Search an Array

```
CR      EQU    13              ;Define return character.
LEN     EQU    132             ;Define line length.
        ORG    $1000           ;Locate LINE at 1000H.
LINE    DS.B   LEN             ;Reserve LEN bytes of storage.
        MOVE.B #LEN-1,D0       ;Initialize D0 to count-1.
        MOVEA.L #LINE,A0       ;A0 gets start address of array.
LOOP    CMPI.B (A0)+,#CR       ;Make the comparison.
        DBEQ    D0,LOOP        ;Double test: if LINE[131-D0]≠13
       <next instruction>      ;  then decr. D0; if D0≠-1 branch
                               ;  to LOOP, else to next inst.
```

- **Program searches an array of bytes to find the first carriage return, ASCII code 13**

Pseudo-Operations in the MC68000 Assembler

- **A *pseudo-operation* is one that is performed by the assembler at *assembly time*, not by the CPU at *run time***
- **EQU defines a symbol to be equal to a constant. Substitution is made *at assemble time***

```
Pi    EQU     3.14
```

- **DS.B (.W or .L) defines a block of storage**
  - **Any label is associated with the first word of the block**

```
Line    DS.B    132
```

  - **The program loader (part of the operating system) accomplishes this**

-more-

Pseudo Operations in the MC68000 Assembler (cont'd.)

- # symbol indicates the value of the symbol instead of a location addressed by the symbol

```
MOVE.L  #1000, D0     ;moves 1000 to D0
MOVE.L    1000, D0    ;moves value at addr. 1000 to D0
```

  - The assembler detects the difference and assembles the appropriate instruction
- ORG specifies a memory address as the origin where the following code will be stored

```
Start          ORG     $4000     ;next instruction/data will be loaded at
                                 ;address 4000H.
```

  - The Motorola assembler uses $ in front of a number to indicate hexadecimal
- Character constants are in single quotes: 'X'

Review of Assembly, Link, Load, and Run Times

- **At *assemble time*, assembly language text is converted to (binary) machine language**
  - **They may be generated by translating instructions, hexadecimal or decimal numbers, characters, etc.**
  - **Addresses are translated by way of a symbol table**
  - **Addresses are adjusted to allow for blocks of memory reserved for arrays, etc.**
- **At *link time*, separately assembled modules are combined and absolute addresses assigned**
- **At *load time*, the binary words are loaded into memory**
- **At r*un time*, the PC is set to the starting address of the loaded module (usually the o.s. makes a jump or procedure call to that address)**

MC68000 Assembly Language Example: Clear a Block

```
MAIN      ...
          MOVE.L    #ARRAY, A0    ;Base of array
          MOVE.W    #COUNT, D0    ;Number of words to clear
          JSR       CLEARW        ;Make the call
          ...
CLEARW    BRA       LOOPE         ;Branch for init. Decr.
LOOPS     CLR.W     (A0)+         ;Autoincrement by 2 .
LOOPE     DBF       D0, LOOPS     ;Dec.D0,fall through if -1
          RTS                     ;Finished.
```

- Subroutine expects block base in A0, count in D0
- Linkage uses the stack pointer, so A7 cannot be used for anything else

Exceptions: Changes to Sequential Instruction Execution

- Exceptions, also called interrupts, cause next instruction fetch from other than PC location
  - Address supplying next instruction called exception vector
- Exceptions can arise from instruction execution, hardware faults, and external conditions
  - Externally generated exceptions usually called interrupts
  - Arithmetic overflow, power failure, I/O operation completion, and out of range memory access are some causes
- A trace bit =1 causes an exception after every instruction
  - Used for debugging purposes

Steps in Handling MC68000 Exceptions

- (1) Status change
  - Temporary copy of status register is made
  - Supervisor mode bit S is set, trace bit T is reset
- (2) Exception vector address is obtained
  - Small address made by shifting 8 bit vector number left 2
  - Contents of the longword at this vector address is the address of the next instruction to be executed
  - The *exception handler* or *interrupt service* routine starts there
- (3) Old PC and status register are pushed onto supervisor stack, addressed by A7' = SSP
- (4) PC is loaded from exception vector address
- Return from handler is done by RTE
  - Like RTR except restores status register instead of CCs

Exception Priorities

- When several exceptions occur at once, which exception vector is used?
- Exceptions have *priorities*, and highest priority exception supplies the vector
- MC68000 allows 7 levels of priority
- Status register contains current priority
- Exceptions with priority ≤ current are ignored

Exceptions and Reset Both Affect Instruction Interpretation

- More processor state needed to describe reset and exception processing

Reset:	Reset input
exc_req:	Single bit exception request
exc_lev⟨2..0⟩:	Exception Level
vect⟨7..0⟩ :	Vector address for this exception
exc := exc_req ∧ (exc_lev⟨2..0⟩ > INT⟨2..0⟩):	There is a request, and the request level is > current mask in status reg.

- exc_lev is the highest priority of any pending exception

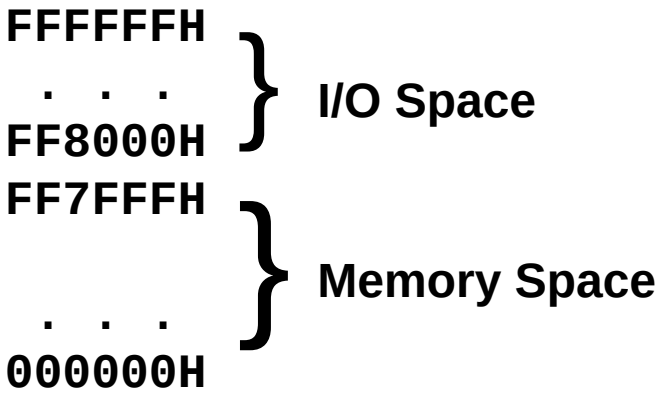
Exceptions Are Sensed Before Fetching Next Instruction

```
Instruction_interpretation := (
Run ∧ ¬(Reset ∨ exc) → (IR ← Mw[PC] : PC ← PC + 2);    Normal execution state
Reset → (INT⟨2..0⟩ ← 7 : S ← 1 : T ← 0:                Machine reset
                SSP ← Ml[0] : PC ← Ml[4] :
                Reset ← 0 : Run ← 1 );
Run ∧ ¬Reset ∧exc → (SSP ← SSP - 4; Ml[SSP] ← PC;      Exception handling
                SSP ← SSP - 2; Mw[SSP] ← Status;
                S ← 1 : T ← 0 : INT⟨2..0⟩ ← exc_lev⟨2..0⟩ :
                PC ← Ml[vect⟨7..0⟩#00₂] );
Instruction_execution                                  ).
```

- Reset starts the computer with a stack pointer from location 0 at the address from location 4

Memory-Mapped I/O

- **No separate I/O space. Part of cpu memory space is devoted/ reserved for I/O instead of RAM or ROM.**
- **Example: MC68000 has a total 24-bit address space. Suppose the top 32K is reserved for I/O:**

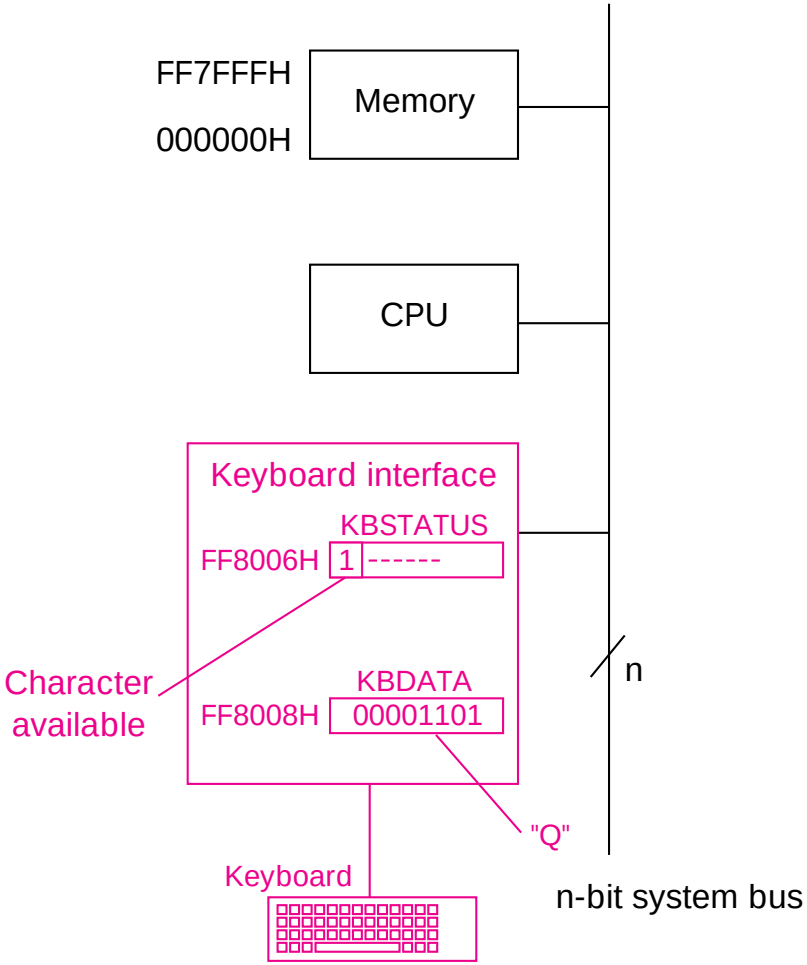


Notice that top 32K can be addressed by a *negative* 16-bit value.

Memory-Mapped I/O in the MC68000

- Memory-mapped I/O allows μprocessor chip to have one bus for both memory and I/O
  - Multiple wires for both address and data
- I/O uses address space that could otherwise contain memory
  - Not popular with machines having limited address bits
- Sizes of I/O and memory "spaces" independent
  - Many or few I/O devices may be installed
  - Much or little memory may be installed
- Spaces are separated by putting I/O at top end of the address space

Fig 3.8 A Memory-Mapped Keyboard Interface



MC68000 has a 24-bit address bus.

Address space runs from 000000H up to FFFFFFH.

A 16-bit address constant can be positive, and sign extend to an address running from 000000H up to the maximum positive value, or negative, and sign extend to an address running from FFFFFFH down to the last negative 16-bit value.

I/O addresses in latter range can be accessed by a 16-bit constant.

The SPARC (Scalable Processor ARChitecture) as a RISC Microprocessor Architecture

- The SPARC is a general register, load-store architecture
- It has only two addressing modes. Address =
  - (Reg + Reg) or (Reg + 31-bit constant)
- Instructions are all 32 bits in length
- SPARC has 69 basic instructions
- Separate floating-point register set
- First implementation had a 4-stage pipeline
- Some important features not inherently RISC
  - Register windows: Separate but overlapping register sets available to calling and called routines
  - 32-bit address, big-endian organization of memory

Fig 3.9 Simplified SPARC Processor State

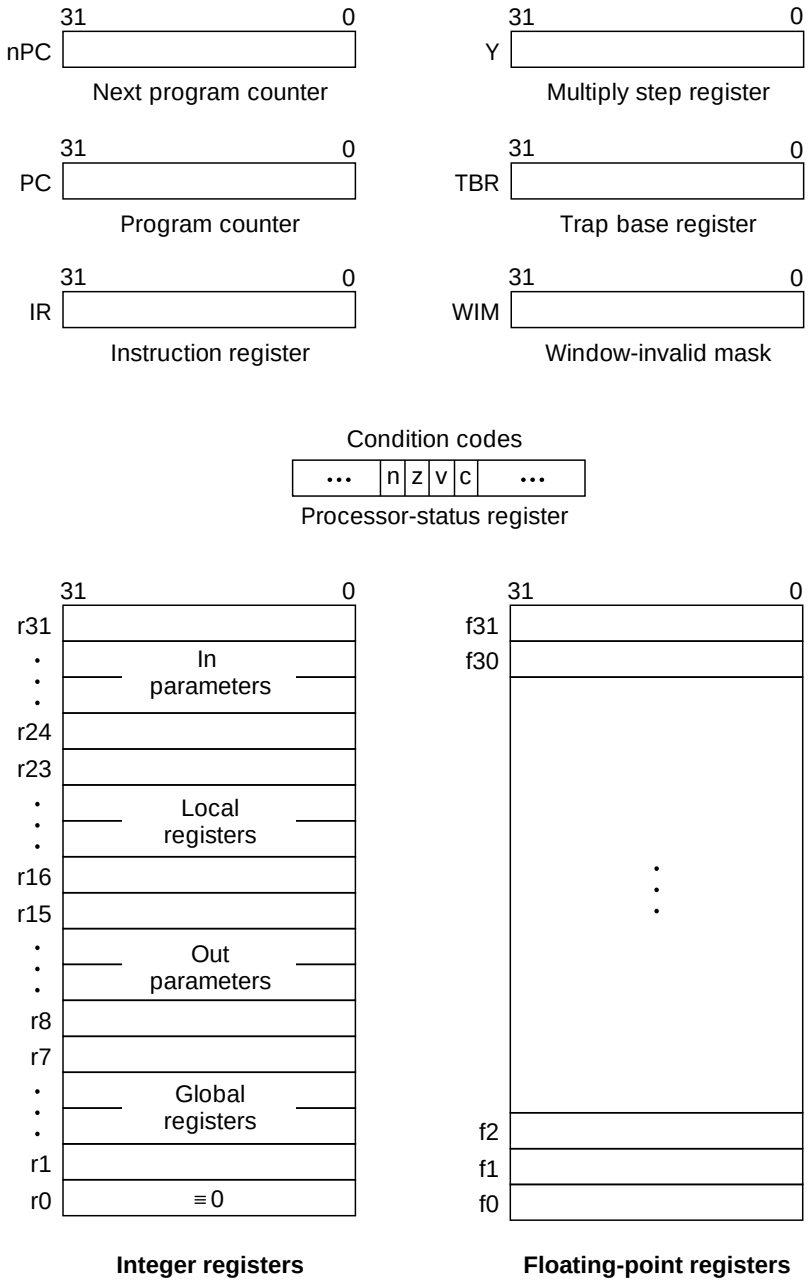
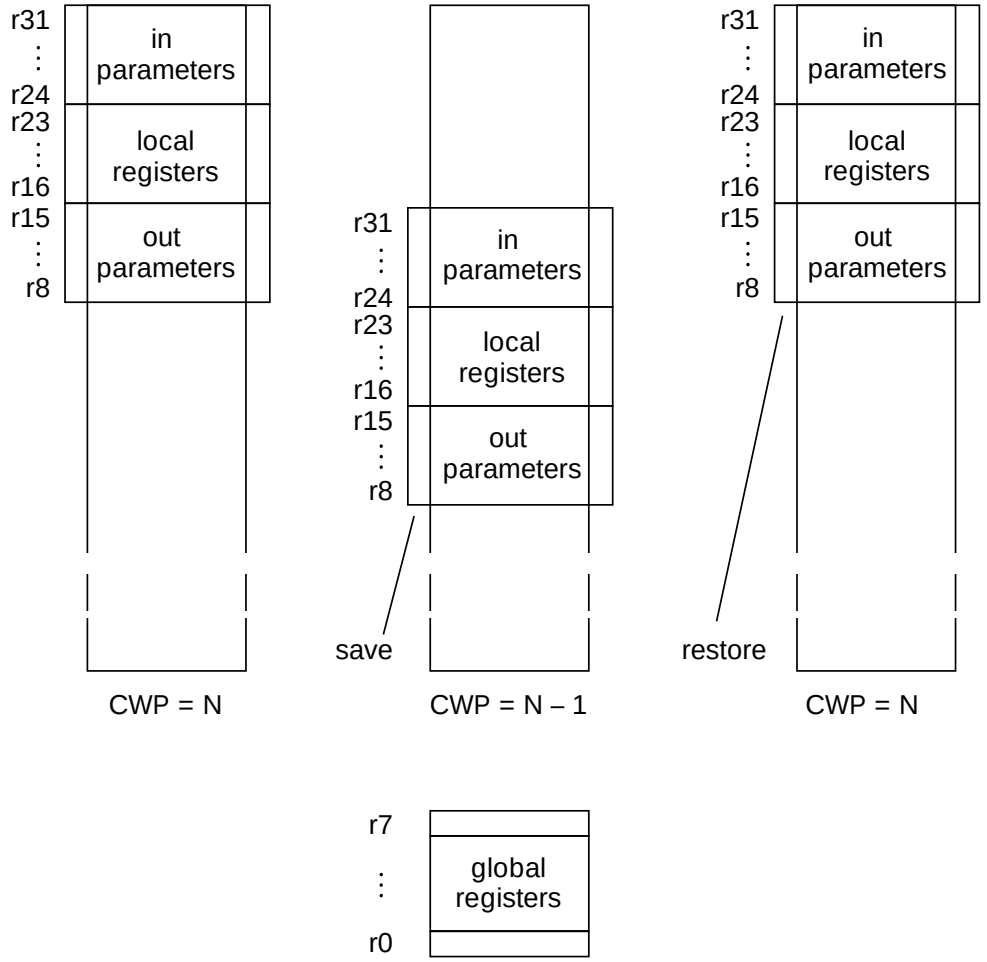


Fig 3.10 SPARC Register Windows Mechanism



SPARC Memory

RTN for the SPARC memory:

Mb[0..2^{32}-1]⟨7..0⟩: **Byte memory**

Mh[a] ⟨15..0⟩ := Mb[a] ⟨7..0⟩#Mb[a + 1] ⟨7..0⟩: **Halfword memory**

M[a] ⟨31..0⟩ := Mh[a] ⟨15..0⟩#Mh[a + 2] ⟨15..0⟩: **Word memory**

Register Windows Format the General Registers

- 32 general integer and address registers are accessible at any one time
  - Global registers G0..G7 are not in any window
  - G0 is always zero: writes to G0 are ignored, reads return 0
  - The other 24 are in a movable window from a total set of 120
- On subroutine call, the starting point changes so that 24–31 before call become 8–15 after
- Registers 8–15 are used for incoming parameters
- Registers 24–31 are for outgoing parameters
- Current Window Pointer CWP locates register 8
- Overflow of register space causes trap

`save`, `restore`, and the Current Window Pointer

- CWP points to the register currently called G8
- `save` moves it to point of the old G24
  - This makes the old G24..G31 into the new G8..G15
- If parameters are placed in G24..G31 by the caller, the callee can get them from G8..G15
- When all windows are used, `save` traps to a routine that saves registers to memory
- Windows wrap around in the available registers
  - Window overflow “spills” the first window and reuses its space

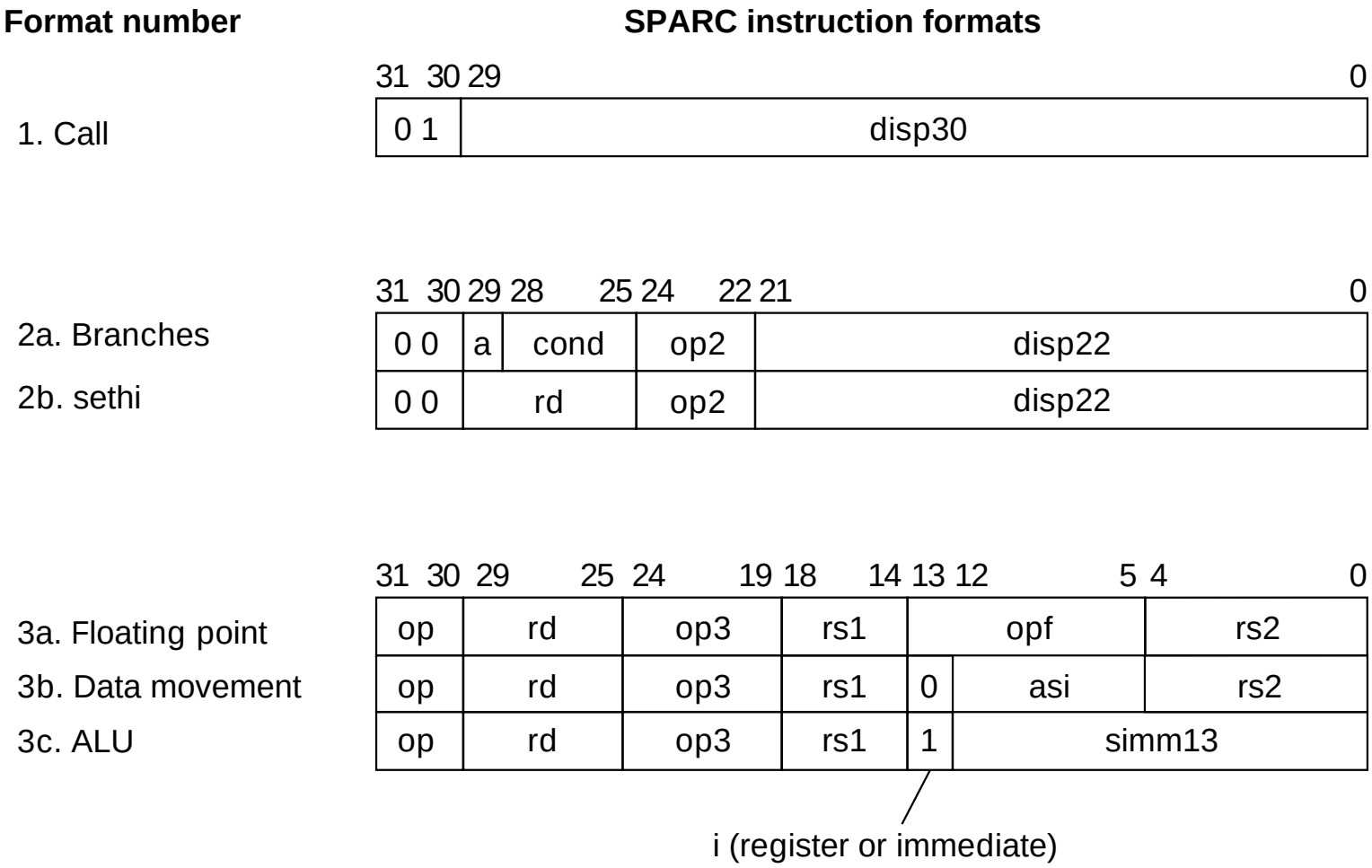
SPARC Operand Addressing

- One mode computes address as sum of 2 registers; G0 gives zero if used
- The other mode adds sign-extended 13-bit constant to a register
- These can serve several purposes
  - Indexed: base in one register, index in another
  - Register indirect: G0 + Gn
  - Displacement: Gn + const, $n \neq 0$
  - Absolute: G0 + constant
- Absolute addressing can only reach the bottom or top 4K bytes of memory

RTN for SPARC Instruction Format

op⟨1..0⟩ := IR⟨31..30⟩:	Instruction class, op code for format 1;
disp30⟨29..0⟩ := IR⟨29..0⟩:	Word displacement for call, format 1;
a := IR⟨29⟩:	Annul bit for branches, format 2a;
cond⟨3..0⟩ := IR⟨28..25⟩:	Branch condition select, format 2a;
rd⟨4..0⟩ := IR⟨29..25⟩:	Destination register for formats 2b & 3;
op2⟨2..0⟩ := IR⟨24..22⟩:	Op code for format 2;
disp22⟨21..0⟩ := IR⟨21..0⟩:	Constant for branch displacement or sethi;
op3⟨5..0⟩ := IR⟨24..19⟩:	Op code for format 3;
rs1⟨4..0⟩ := IR⟨18..14⟩:	Source register 1 for format 3;
opf⟨8..0⟩ := IR⟨13..5⟩:	Sub-op code for floating point, format 3a;
i := IR⟨13⟩:	Immediate operand indicator, formats 3b & c;
simm13⟨12..0⟩ := IR⟨12..0⟩:	Signed immediate operand for format 3c;
rs2⟨4..0⟩ := IR⟨4..0⟩:	Source register 2 for format 3b.

Fig 3.11 SPARC Instruction Formats



- **Three basic formats with variations**

RTN For SPARC Addressing Modes

adr⟨31..0⟩ := (i=0 → r[rs1] + r[rs2]: i=1 → r[rs1] + simm13⟨12..0⟩ {sign ext.}):	Address for load, store, and jump
calladr⟨31..0⟩ := PC⟨31..0⟩ + disp30⟨29..0⟩ #002:	Call relative address
bradr⟨31..0⟩ := PC⟨31..0⟩ + disp22⟨21..0⟩ #002{sign ext.}:	Branch address

RTN For SPARC Instruction Interpretation

instruction_interpretation := (IR ← M[PC]; instruction_execution;
update_PC_and_nPC; instruction_interpretation):

Tbl 3.8 SPARC Data Movement Instructions

Inst.	Op.	OPCODE	Meaning
ldsb	11	00 1001	Load signed byte
ldsh	11	00 1010	Load signed halfword
ldsw	11	00 1000	Load signed word
ldub	11	00 0001	Load unsigned byte
lduh	11	00 0010	Load unsigned halfword
ldd	11	00 0011	Load doubleword
stb	11	00 0101	Store byte
sth	11	00 0110	Store halfword
stw	11	00 0100	Store word
std	11	00 0111	Store double word
swap	11	00 1111	Swap register with memory
or	10	00 0010	r[d] ← r[s1] OR (r[rs2] or immediate)
sethi	00	Op2=100	High order 22 bits of Rdst ← disp22

Register and Immediate Moves in the SPARC

- OR is used with a G0 operand to do register-to-register moves
- To load a register with a 32-bit constant, a 2-instruction sequence is used

```
SETHI  R17, #upper22
OR     R17, R17, #lower10
```

- Doublewords are loaded into an even register and the next higher odd one
- Floating-point instructions are not covered, but the 32 FP registers can hold single-length numbers, or 16 64-bit FP, or 8 128-bit FP numbers

Tbl 3.9 SPARC Arithmetic Instructions

Inst.	Op.	OPCODE	Meaning
add	10	0S 0000	Add or add and set condition codes
addx	10	0S 1000	Add with carry: set CCs or not
sub	10	0S 0100	Subtract: subtract and set CCs or not
subx	10	0S 1100	Subtract with borrow: set CCs or not
mulscc	10	10 1100	Do one step of multiply

- All are format 3, Op = 10
- CCs are set if S = 1 and not if S = 0
- Both register and immediate forms are available
- Multiply is done by software using MULSCC or using floating-point instructions
  - Multiply is hard to do in one clock but multiply step is not

Tbl 3.10 SPARC Logical and Shift Instructions

Inst.	Op.	OPCODE	Meaning
AND	10	0S 0001	AND, set CCs if S=1 or not if S=0
ANDN	10	0S 0101	NAND, set CCs or not
OR	10	0S 0010	OR, set CCs or not
ORN	10	0S 0110	NOR, set CCs or not
XOR	10	0S 0011	XNOR(Equiv), set CCs or not
SLL	10	10 0101	Shift left logical, count in RSRC2 or imm13
SRL	10	10 0110	Shift right logical, count in RSRC2 or imm13
SRA	10	10 0111	Shift right arithmetic, count as above

- All instructions use format 3 with op = 10
  - Both register and immediate forms are available
- Condition codes set if S = 1 and undisturbed if S = 0

Tbl 3.11 SPARC Branch and Control Transfer Instructions

Inst.	Format	Op	Op2 or Op3	Meaning
ba	2	00	010	Unconditional branch
bcc	2	00	010	Conditional branch
call	1	01		Call & save PC in R15
jmpl	3	10	11 1000	Jmp to EA, save PC in Rdst
save	3	10	11 1100	New register window, & ADD
restore	3	10	11 1101	Restore reg. window, & ADD

Some condition fields:

Inst.	COND	Inst.	COND	Inst.	COND	Inst.	COND
ba	1000	bne	1001	be	0001	ble	0010
bcc	1101	bcs	0101	bneg	0110	bvc	1111
bvs	0111						

Fig 3.12 Example SPARC Assembly Program

```
        .begin
        .org
prog:   ld      [x],    %r1         ! Load a word from M[x] into register %r1.
        ld      [y],    %r2         ! Load a word from M[y] into register %r2.
        addcc%r1, %r2, %r3          ! %r3 ← %r1 + %r2  ; set CCs.
        st      %r3,    [z]         ! Store sum into M[z].
        jmpl    %r15,   +8, %r0     ! Return to caller.
        nop                         ! Branch delay slot.
x:      15                          ! Reserve storage for x, y, and z.
y:      9
z:      0
        .end
```

Note different syntax for SPARC.
Note r15 contains return address—placed there by the OS in this case.

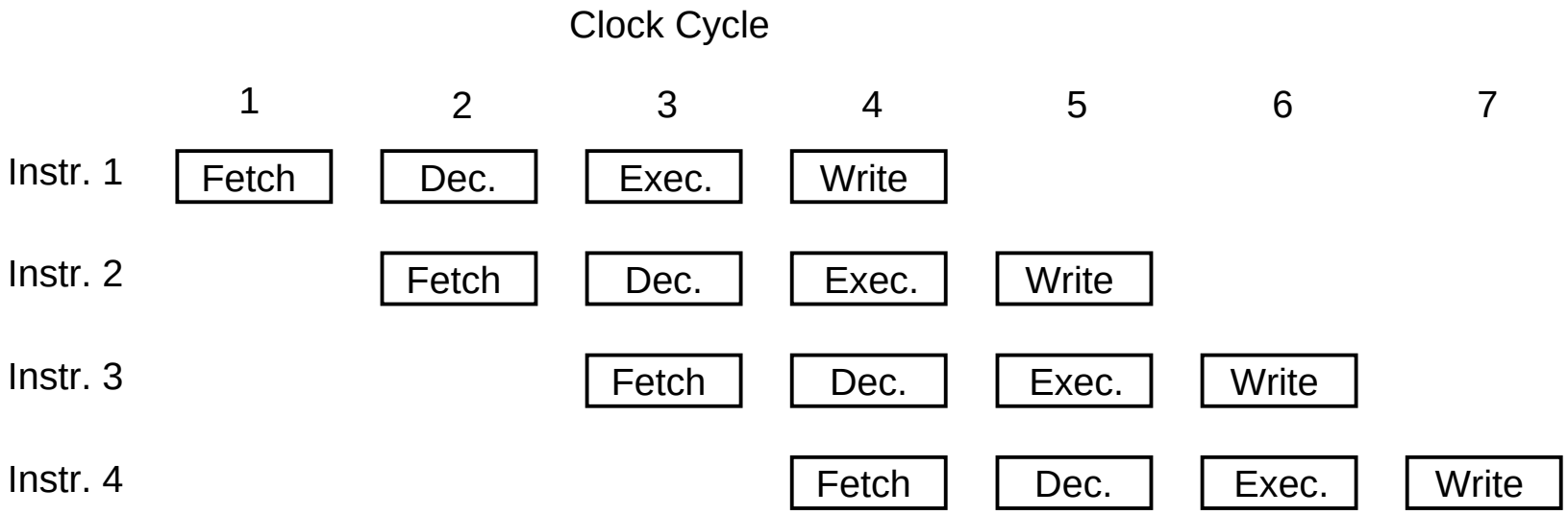
Fig 3.13 Example of Subroutine Linkage in the SPARC

```
        .begin
        .org
prog:   ld      [x], %o0        !Pass parameters in
        ld      [y], %o1        !  first 3 output registers.
        call    add3            !Call subroutine to put result in %o0.
        mov     -17, %o2        !Set last parameter in delay slot
        st      %o0, [z]        !Store returned result.
        ...
x:      15
y:      9
z:      0
add3:   save    %sp,-(16*4),%sp !Get new window and adjust stack pointer.
        add     %i0, %i1, %l0   !Add parameters that now appear in
        add     %l0, %i3, %l0   !  input registers using a local.
        ret                     !Return. Short for jmp %i7+8.
        restore %l0, 0, %o0     !Result moved to caller's %o0.
        .end
```

Pipelining of the SPARC Architecture

- Many aspects of the SPARC design are in support of a pipelined implementation
  - Simple addressing modes, simple instructions, delayed branches, load-store architecture
- Simplest form of pipelining is fetch-execute overlap—fetching next instruction while executing current instruction
- Pipelining breaks instruction processing into steps
  - A step of one instruction overlaps different steps for others
- A new instruction is started (issued) before previously issued instructions are complete
- Instructions guaranteed to complete in order

Fig 3.14 The SPARC MB86900 Pipeline



- **4 pipeline stages are Fetch, Decode, Execute, and Write**
- **Results are written to registers in Write stage**

Pipeline Hazards

- Will be discussed later, but main issue is:
- Branch or jump change the PC as late as Exec or Write, but next instruction has already been fetched
  - One solution is delayed branch
  - One (maybe 2) instruction following branch is always executed, regardless of whether branch is taken
  - SPARC has a delayed branch with one delay slot, but also allows the delay slot instruction to be annulled (have no effect on the machine state) if the branch is not taken
- Registers to be written by one instruction may be needed by another already in the pipeline, before the update has happened (data hazard)

CISC versus RISC: Recap

- CISCs supply powerful instructions tailored to commonly used operations, stack operations, subroutine linkage, etc.
- RISCs require more instructions to do the same job
- CISC instructions take varying lengths of time
- RISC instructions can all be executed in the same few-cycle pipeline
- RISCs should be able to finish (nearly) one instruction per clock cycle

Key Concepts: RISC versus CISC

- **While a RISC machine may possibly have fewer instructions than a CISC, the instructions are always simpler. Multistep arithmetic operations are confined to special units.**
- **Like all RISCs, the SPARC is a load-store machine. Arithmetic operates only on values in registers.**
- **A few regular instruction formats and limited addressing modes make instruction decode and operand determination fast.**
- **Branch delays are quite typical of RISC machines and arise from the way a pipeline processes branch instructions.**
- **The SPARC does not have a load delay, which some RISCs do, and does have register windows, which many RISCs do not.**

Chapter 3 Summary

- Machine price/performance are the driving forces.
  - Performance can be measured in many ways: MIPS, execution time, Whetstone, Dhrystone, SPEC benchmarks.
- CISC machines have fewer instructions that do more.
  - Instruction word length may vary widely
  - Addressing modes encourage memory traffic
  - CISC instructions are hard to map onto modern architectures
- RISC machines usually have
  - One word per instruction
  - Load/store memory access
  - Simple instructions and addressing modes
  - Result in allowing higher clock cycles, prefetching, etc.