Chapter 3 Topics

C
S
D
A
2/e

3.1   Machine characteristics and performance

3.2   RISC vs. CISC

3.3   A CISC microprocessor: The Motorola MC68000

3.4   The SPARC: a RISC architecture

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

Quantitative Performance Measurement

Consider two auto routes, the old one, which allowed an average speed of 34 mph, and the new one, which permitted 46 mph. What is the speedup of the new one over the old one?

Conventionally the speedup is calculated as follows:

$$Speedup = \frac{SpeedOnNewRoute}{SpeedOnOldRoute} = \frac{S_{new}}{S_{old}} = \frac{46}{34} = 1.35$$

For a speedup of 0.35, or 35%.
Alternately, the % speedup can be calculated directly:

$$\%Speedup = \frac{S_{new} - S_{old}}{S_{old}} \times 100 = \frac{46 - 34}{34} \times 100 = \frac{12}{34} \times 100 = 35\%$$

Quantitative Performance Measurement

Many measurements are in terms of the time, T, it takes to accomplish some task. Recall that Time, T, is the reciprocal of Speed, S= 1/T. If the improvement is measured by recording travel time rather than travel speed the equation changes as follows:

$$Speedup = \frac{S_{new}}{S_{old}} = \frac{\frac{1}{T_{new}}}{\frac{1}{T_{old}}} = \frac{T_{old}}{T_{new}} = \frac{96}{71} = 1.35, \text{ or } 35\%$$

Once again, the % speedup can be calculated directly:

$$\%Speedup = \frac{T_{old} - T_{new}}{T_{new}} \times 100 = \frac{96 - 71}{71} \times 100 = \frac{25}{71} \times 100 = 35\%$$

A Classic Example

CLASSIC EXAMPLE: ESTIMATING PERFORMANCE A certain computer system takes 125 ms to render a certain graphic image, and this time is reduced to 100 ms when a graphics processor card is added to the system. What is the speedup?

$$\text{Speedup} = \frac{T_{old}}{T_{new}} = \frac{125}{100} = 1.25, \text{ or a 25\% speedup}$$

Getting Finer-Grained

- The execution time can be calculated from the count of how many instructions have executed, IC, the average number of clock cycles per instruction, CPI, and the clock period, τ.

- This is an important equation that will be used throughout the text.

$$\text{Execution time } = T = IC \times CPI \times \tau$$

Example 3.1 Speedup Due to a Clock Frequency Increase The master clock in a certain computer system is increased in frequency from 700 MHz to 1.2 GHz. What is the speedup due to this improvement if no other factors such as memory access time interfere with the improvement?

Since, according to the problem definition, neither IC nor CPI changed, and since the clock period, τ, is proportional to the reciprocal of clock frequency,

$$\text{Speedup } = \frac{(IC \times CPI \times \tau)_{old}}{(IC \times CPI \times \tau)_{new}} = \frac{1/700}{1/1200} = \frac{1200}{700} = 1.71, \text{ or } 71\% \text{ speedup}$$

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 & 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 Developing an Instruction Set Architecture

- *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 MC68000 Programmer's Model
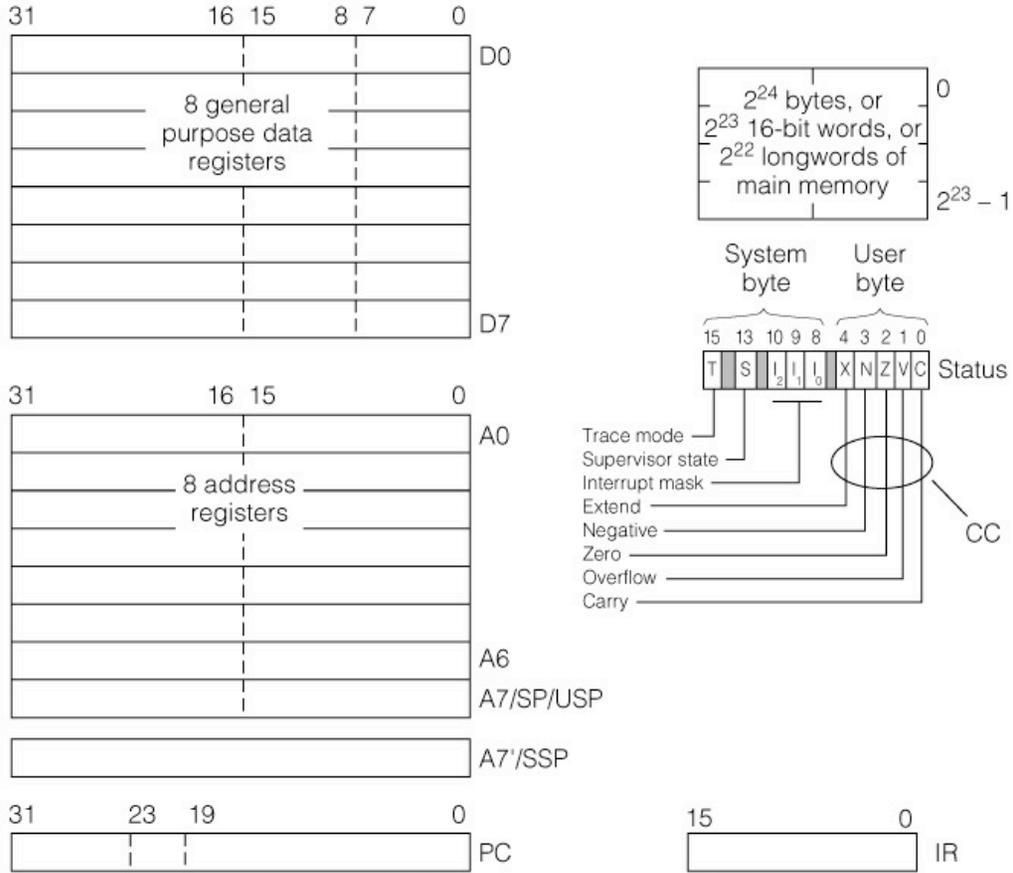
Copyright © 2004 Pearson Prentice Hall, Inc.

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 & User bytes

  - Arithmetic status (N, Z, V, C, X) is in user status byte

  - System status has Supervisor & 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⟨23..0⟩:	Program counter in original MC68000
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:

$Mb[0..2^{24}-1]\langle 7..0 \rangle$:   Memory as bytes

$Mw[ad]\langle 15..0 \rangle := Mb[ad]\#Mb[ad+1]$:   Memory as words

$Ml[ad]\langle 31..0 \rangle := Mw[ad]\#Mw[ad+2]$:   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.
  - The default, ADD, for example, is Word operands.
  - Operand length is encoded into 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

15   0

IR | op | rg2 | md2 | md1 | rg1 |
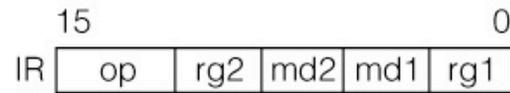
(a) A 1-word move instruction

15   0

··· | md1 | rg1 | IR
16-bit constant | Extra word

(b) A 2-word instruction

15   0

IR | ··· | md1 | rg1 |
Extra word | 16-bit constant
Extra word | 16-bit constant

(c) A 3-word instruction

15   0

··· | 110 | Reg | IR
d/a | Index reg | w/l | 000 | disp8 | Extra word
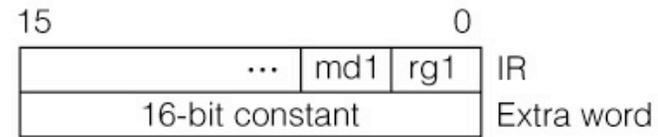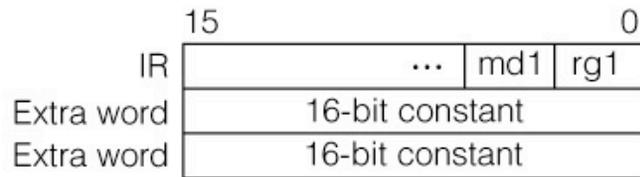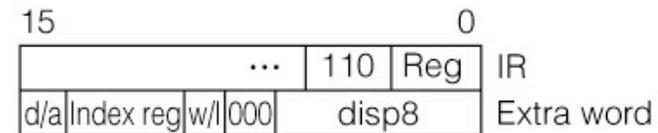
(d) Instruction with indexed address

Copyright © 2004 Pearson Prentice Hall, Inc.

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

5	4	3	2	1	0
mode			reg		
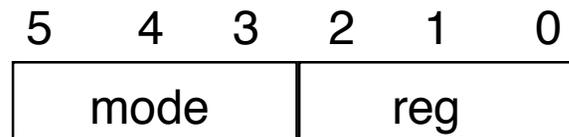
Provides access paths to operands

- 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 inst.

- Exception: when specifying the destination of a MOVE instruction the mode and reg fields are reversed.

```
 5   4   3   2   1   0
┌──────────────┬──────────────┐
│     mode     │     reg      │
└──────────────┴──────────────┘
```
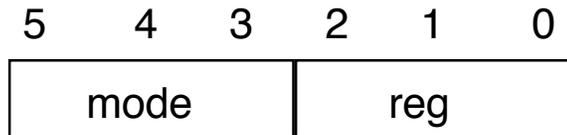
MC68000 Addressing Modes

Name	Mode	Reg.	Assembler Syntax	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

	5	4	3	2	1	0
	mode			reg		
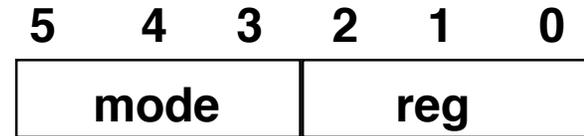
- The addressing modes interpret many items
  - The instruction: in the IR register
  - The following 16 bit word: described as Mw[PC]
  - The D and A registers in the CPU
- Many addressing modes calculate an effective memory address
- Some modes designate a register
- Some modes result in a constant operand
- There are restrictions on the use of some modes

RTN Formatting for Effective Address Calculation

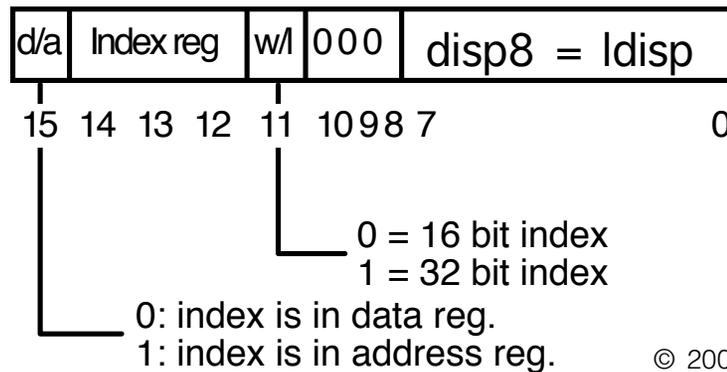$XR[0..15]\langle 31..0 \rangle :=$

     $D[0..7]\langle 31..0 \rangle \# A[0..7]\langle 31..0 \rangle$:　　　Index register can be D or A;

$xr\langle 3..0 \rangle := Mw[PC]\langle 15..12 \rangle$:　　　　　Index specifier for index mode;

$wl := Mw[PC]\langle 11 \rangle$:　　　　　　　　　Short or long index flag;

$dsp8\langle 7..0 \rangle := Mw[PC]\langle 7..0 \rangle$:　　　　Displacement for index mode;

$index := (\ (wl=0) \rightarrow XR[xr]\langle 15..0 \rangle$:　　Short or

        $(wl=1) \rightarrow XR[xr]\langle 31..0 \rangle)$:　　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

d/a	Index reg	w/l	0 0 0	disp8 = ldisp

15　14　13　12　11　10 9 8　7　　　　　　0

0 = 16 bit index
1 = 32 bit index

0: index is in data reg.
1: index is in address reg.

© 2004 Prentice Hall

Modes That Calculate a Memory Address Using a Register

5	4	3	2	1	0
010 - 110			000 - 111		
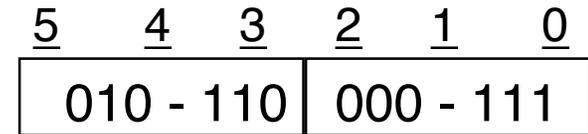
- md and rg are the 3-bit mode and reg. fields.
- ea stands for *effective address*
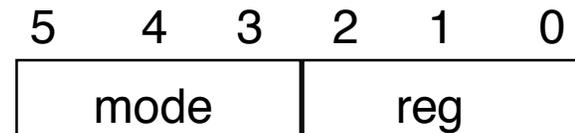
5	4	3	2	1	0
mode			reg		

ea(md, rg) := (
  (md = 2) → A[rg⟨2..0⟩]:   Mode 2 is 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 reg Field to Expand the Number of Modes

5	4	3	2	1	0
1	1	1	reg		

- 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 Mode 2: Address Register Indirect

5	4	3	2	1	0
0	1	0		**reg**	

68000 Registers

Main memory

Address register indirect

...	010	Reg

A0

...

Address

A7

Ex: MOVE (A6), ...

Operand

- 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

5	4	3	2	1	0
1	1	0		**reg**	

Mode 6: Based indexed addressing

68000 Registers

Main memory

		...	110	Reg
d/a	Index reg	w/l	000	disp8 = ldisp

15 14 13 12 11 1098 7   0

A0

...

Base address •

A7

0 = 16 bit index
1 = 32 bit index

0: index is in data reg.
1: index is in address reg.

Ex: `MOVE.W LDISP (A6, D4), ...`

+

Operand

Index (16 or 32) •   D0-D7
                     A0-A7
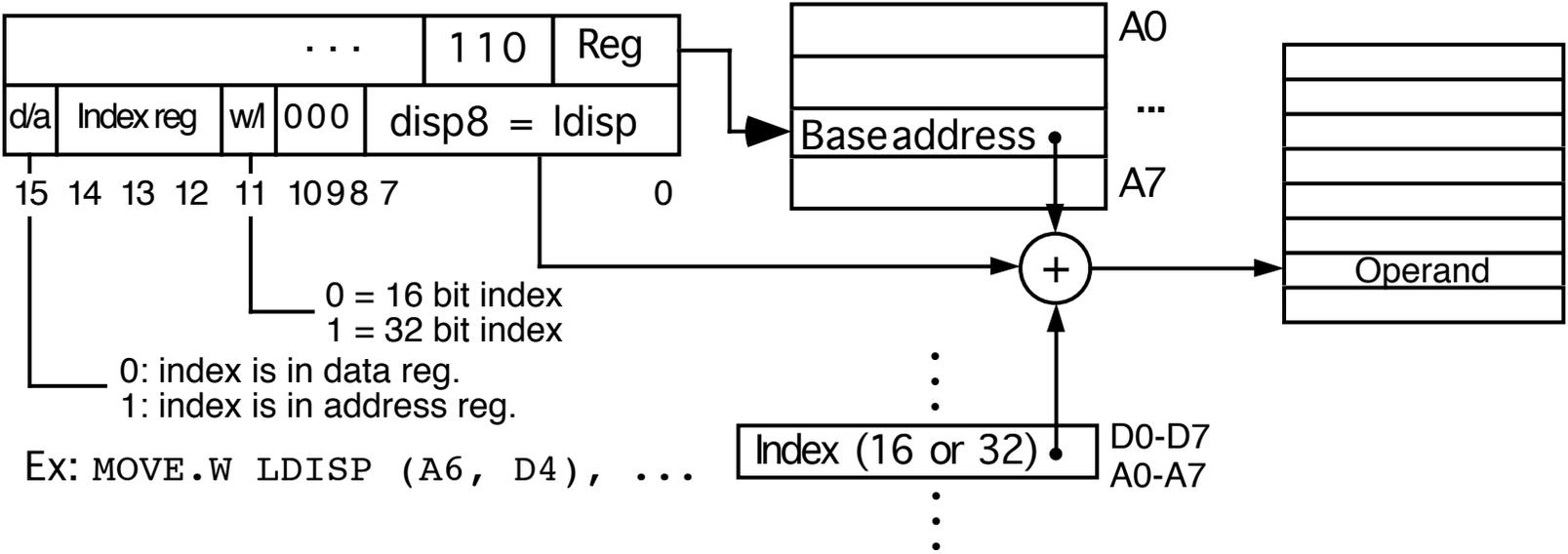
- **Three things are added to get the address**

Modes 7-0 and 7-1: Absolute Addressing

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

Main memory

Absolute short addressing

...	111	000
addr16		

15   0

Ex: MOVE.B PRINTERPORT.W, ...

(Sign extend to 32-bits)

Operand

Absolute long addressing

...	111	001
addr32Hi		
addr32Lo		

15   0

Ex: MOVE.W INTVECT.L, ...

Concat.

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

Mode 7-3 Relative Indexed Addressing

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

Relative indexed addressing

	...	1 1 1	0 1 1
d/a	Index reg	w/l	000	disp8 = ldisp

15 14 13 12 11 1098 7   0

Program counter

Main memory

+

Operand

0 = 16 bit index
1 = 32 bit index

0: index is in data reg.
1: index is in address reg.

Ex: `MOVE.W LDISP (PC, D4), ...`

Index (16 or 32)   D0-D7
                    A0-A7
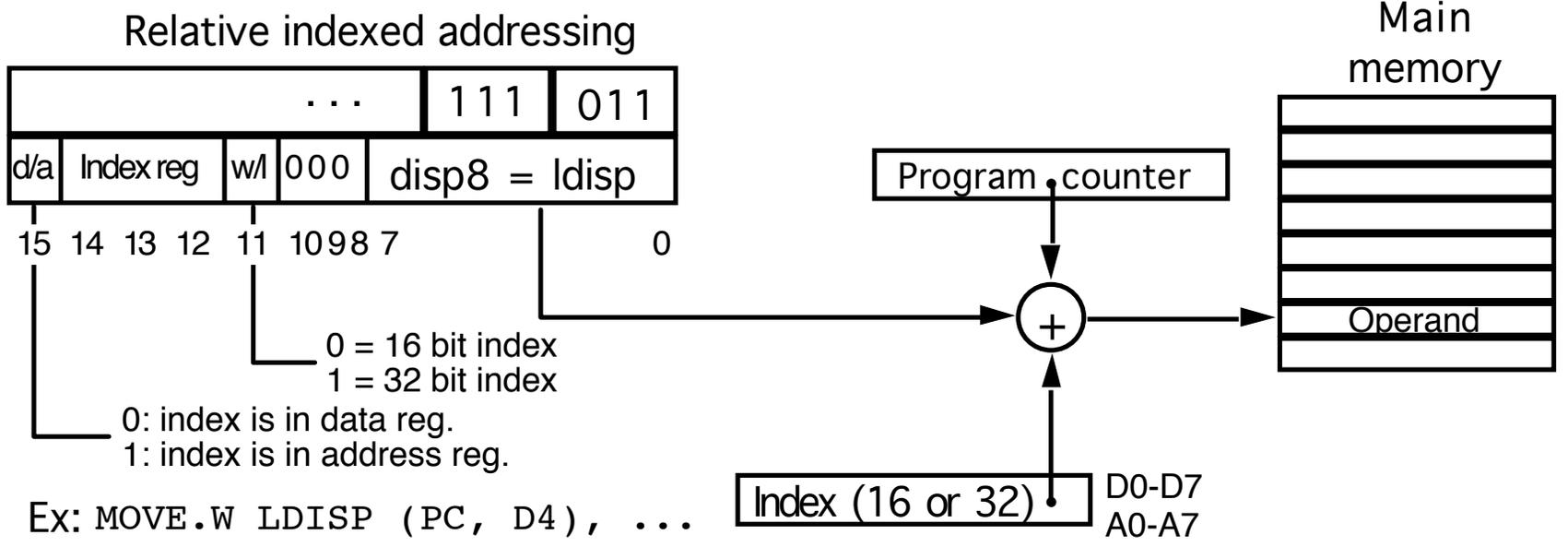
- Same as indexed mode but uses PC instead of A register as base

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
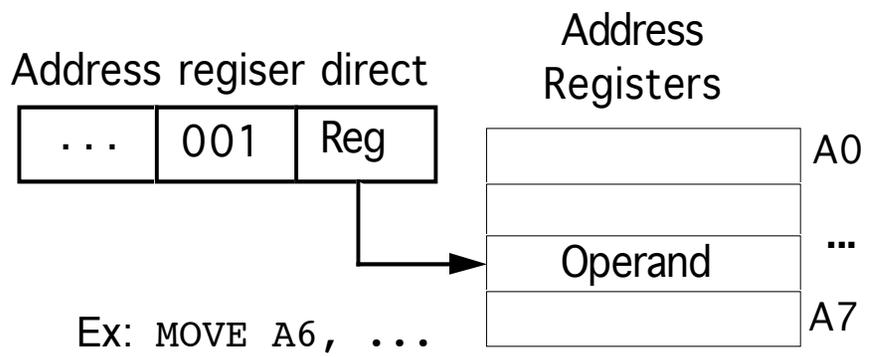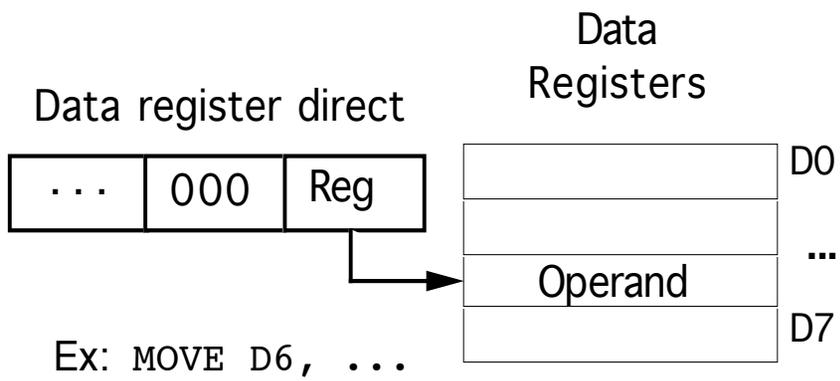   which to use.
A long operand can be

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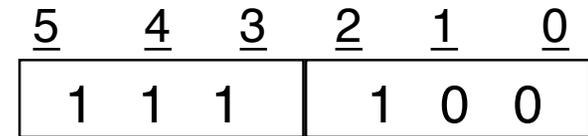
5	4	3	2	1	0
0	0	0 (D)		**reg**	
0	0	1 (A)			

Data register direct

Data Registers

...	000	Reg

	D0
Operand	...
	D7

Ex: `MOVE D6, ...`

Address regiser direct

Address Registers

...	001	Reg

	A0
Operand	...
	A7

Ex: `MOVE A6, ...`
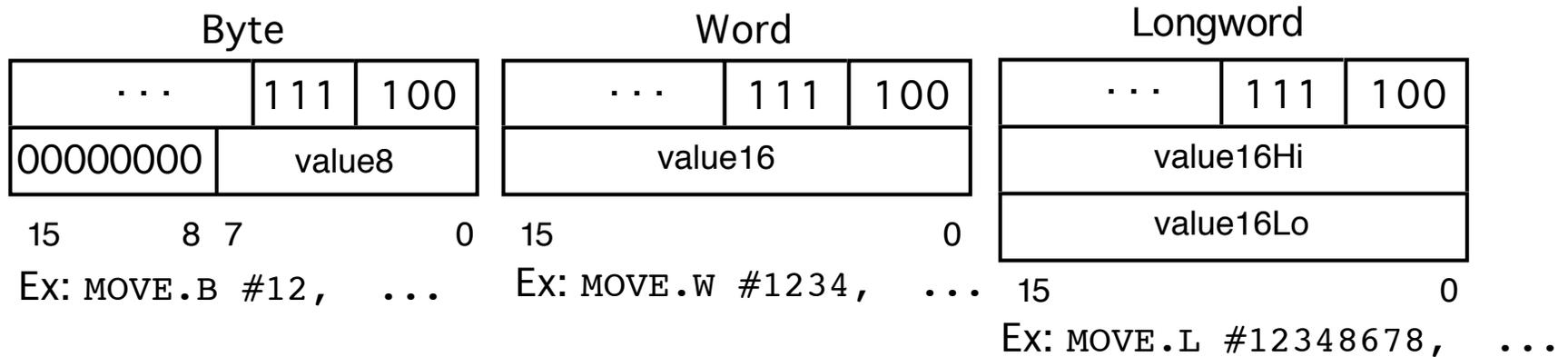
- The register itself provides a place to store a result or a place to get an operand
- There is no memory address with this mode

Fig. 3.5 Mod 7-4: 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

Byte

...	111	100
00000000	value8	

```
15        8 7              0
```

Ex: `MOVE.B #12, ...`

Word

...	111	100
value16		

```
15                        0
```

Ex: `MOVE.W #1234, ...`

Longword

...	111	100
value16Hi		
value16Lo		

```
15                        0
```

Ex: `MOVE.L #12348678, ...`

- 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) ∧ ¬(md=7 ∧(rg=2∨rg=3)):

- The MC68000 disallows relative addressing (md7 rg 2 or 3) 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

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 Data Movement Instructions in the MC68000

Inst.	Operands	First Instruction Word	X	N	Z	V	C	Operation	Size
MOVE.B	EAs, EAd	0001dddddd ssssss	–	x	x	0	0	dst ← src	byte
MOVE.W	EAs, EAd	0011dddddd ssssss	–	x	x	0	0	dst ← src	word
MOVE.L	EAs, EAd	0010dddddd ssssss	–	x	x	0	0	dst ← src	long
MOVEA.W	EAs, An	0011rrr001 ssssss	–	–	–	–	–	An ← src	word
MOVEA.L	EAs, An	0010rrr001 ssssss	–	–	–	–	–	An ← src	long
LEA.L	EAc, An	0100aaa111 ssssss	–	–	–	–	–	An ← EA	address
EXG	Dx, Dy	1100xxx101000yyy	–	–	–	–	–	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

  $\text{op}\langle 3..0\rangle := \text{IR}\langle 15..12\rangle$: $\text{rg1}\langle 2..0\rangle := \text{IR}\langle 2..0\rangle$: $\text{md1}\langle 2..0\rangle := \text{IR}\langle 5..3\rangle$:
  $\text{rg2}\langle 2..0\rangle := \text{IR}\langle 11..9\rangle$: $\text{md2}\langle 2..0\rangle := \text{IR}\langle 8..6\rangle$:

  $\text{tmp}\langle 31..0\rangle$:
  $\text{move } (:= \text{op}\langle 3..2\rangle := 0) \rightarrow ($
  $\qquad \text{tmp} \leftarrow \text{opnd}(\text{md1, rg1})$;
  $\qquad (\ Z \leftarrow (\text{tmp}=0): N \leftarrow (\text{tmp}<0): V \leftarrow 0: C \leftarrow 0\ )$:
  $\qquad \text{rslt}(\text{md2, rg2}) \leftarrow \text{tmp}$):

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

MC68000 Integer Arithmetic and Logic Instructions

Op.	Operands	Inst. word	X	N	Z	V	C	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	1011rrrxxxaaaaaa	–	x	x	x	x	dst-src	b, w, l
CMPI	#dat,EA	00001100wwaaaaaa	–	x	x	x	x	dst-imm.data	b, w, l
MULS	EA, Dn	1100rrr111aaaaaa	–	x	x	0	0	Dn←Dn*src	l←w*w
MULU	EA,Dn	1100rrr011aaaaaa	–	x	x	0	0	Dn←Dn*src	l←w*w
DIVS	EA,Dn	1000rrr111aaaaaa	–	x	x	x	0	Dn←Dn/src	l←l/w
DIVU	EA,Dn	1000rrr011aaaaaa	–	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	1011rrrwwaaaaaa	–	x	x	0	0	dst←dst⊕src	b, w, l
CLR	EAs	01000010wwaaaaaa	–	0	1	0	0	dst←0	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

aaaaaa is the 6-bit addressing mode specifier mmmrrr

www: B—100, W—101, L—110

xxx: B—000, W—001, L—010

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

op   Dn   tbl abv.   EA
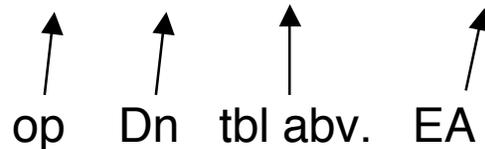
Note: There are several exceptions to the rule above. See text and Mfr. Data sheet.

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

$tmp\langle 31..0\rangle$:   temporary register for address

sub (:= op=9) $\rightarrow$ (
  $(md2\langle 2\rangle =0) \rightarrow D[rg2] \leftarrow D[rg2] - opnd(md1, rg1)$:
  $(md2\langle 2\rangle =1) \rightarrow (memval(md1, rg1) \rightarrow (tmp \leftarrow ea(md1, rg1)$;
                                    $M[tmp] \leftarrow M[tmp] - D[rg2]$):
      $\neg memval(md1, rg1) \rightarrow rslt(md1, rg1) \leftarrow rslt(md1, rg1) - D[rg2])$
                                    ):

- This definition does not handle the condition codes

MC68000 Arithmetic Shifts and Single Word Rotates

Op.	Operands	Inst. word	XNZVC
ASd	EA	1110000d11aaaaaa	xxxxx
ASd	#cnt,Dn	1110cccdww000rrr	xxxxx
ASd	Dm,Dn	1110RRRdww100rrr	xxxxx
ROd	EA	1110011d11aaaaaa	-xx0x
ROd	#cnt,Dn	1110cccdww011rrr	-xx0x
ROd	Dm,Dn	1110RRRdww111rrr	-xx0x
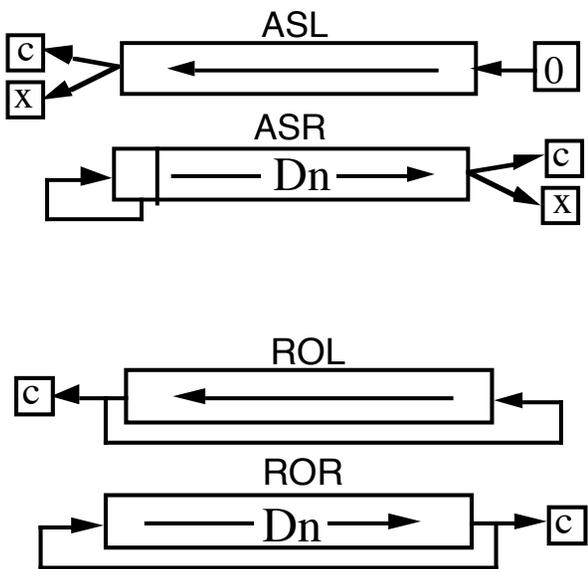
- d is L or R for left or right shift, respectively
- EA form has shift count of 1
- ww is word size: 00–Byte, 01–Word, 10–Long Word

MC68000 Logical Shifts and Extended Rotates

Op.	Operands	Inst. word	XNZVC
LSd	EA	1110001d11aaaaaa	xxx0x
LSd	#cnt,Dn	1110cccdww001rrr	xxx0x
LSd	Dm,Dn	1110RRRdww101rrr	xxx0x

Op.	Operands	Inst. word	XNZVC
ROXd	EA	1110010d11aaaaaa	xxx0x
ROXd	#cnt,Dn	1110cccdww010rrr	xxx0x
ROXd	Dm,Dn	1110RRRdww110rrr	xxx0x
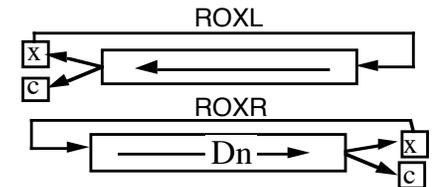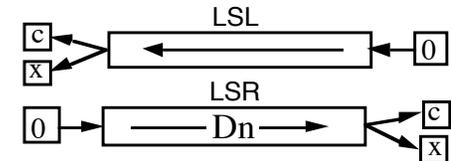
- Field ww specifies byte, word, or longword
- N & Z set according to result, C= last bit shifted out

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`

- disp is *dddddddd* unless *dddddddd* =0, in which case it is contained in the extra word DDDDDDDDDDDDDDDD
- DBcc is used for counted loops with an optional end condition.
  - "Decrement and branch until cond."
- 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	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	$C + Z$
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	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
1111	less or equal	LE	$N \cdot \overline{V} + \overline{N} \cdot V + Z$
0010[†]	high	HI	$\overline{C} \cdot \overline{Z}$
1000	overflow clear	VC	$\overline{V}$
1001	overflow set	VS	V

[†]Assumes unsigned operands

```
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

Figure 3.6 Example Program to 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-

C
S
D
A
2/e

- #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

C S D A 2/e

- 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 & absolute addresses assigned
- At *load time*, the binary words are loaded into memory
- At *run 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 reg. 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$\langle 2..0 \rangle$:   Exception Level
vect$\langle 7..0 \rangle$:   Vector address for this exception
exc := exc_req $\wedge$ (exc_lev$\langle 2..0 \rangle$ > INT$\langle 2..0 \rangle$): 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 ← MI[0] : PC ← MI[4] :
                Reset ← 0 : Run ← 1);
Run ∧ ¬Reset ∧exc → (SSP ← SSP - 4; MI[SSP] ← PC;   Exception handling
                SSP ← SSP - 2; Mw[SSP] ← Status;
                S ← 1 : T ← 0 : INT⟨2..0⟩ ← exc_lev⟨2..0⟩ :
                PC ← MI[vect⟨7..0⟩#00$_2$]);
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:

```
FFFFFFH
 . . .        } I/O Space
FF8000H

FF7FFFH
              } Memory Space
 . . .
000000H
```
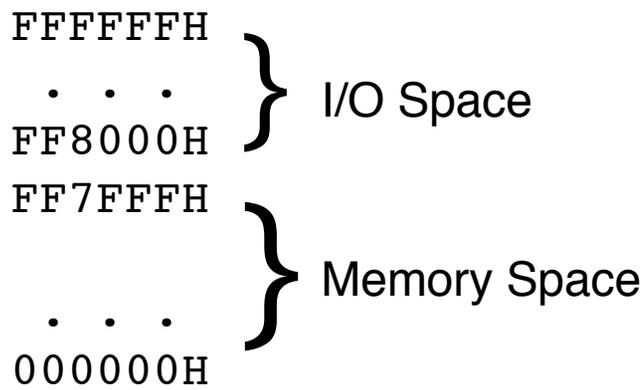
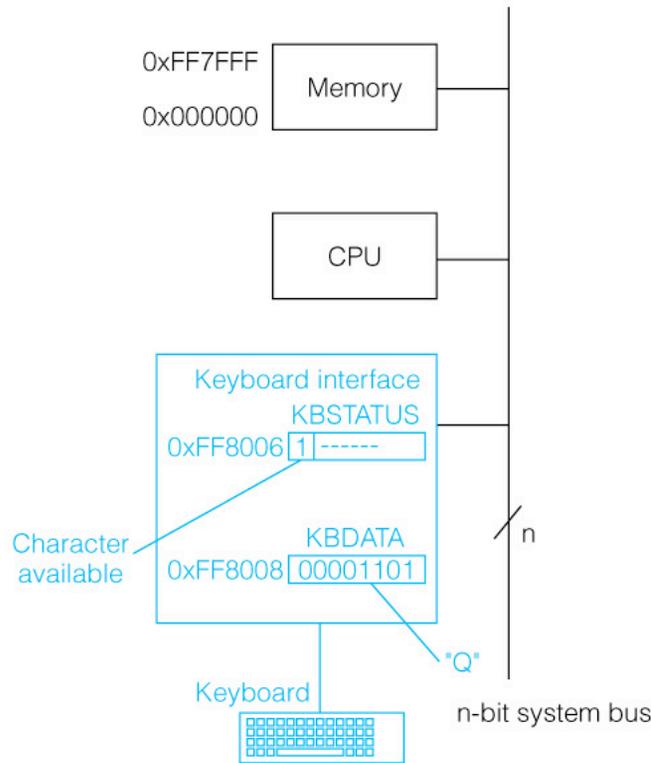Notice that top 32K can be addressed by a *negative* 16-bit value.

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 & 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

0xFF7FFF

Memory

0x000000

CPU

Keyboard interface

KBSTATUS

0xFF8006 | 1 | ------ |

Character available

KBDATA

0xFF8008 | 00001101 |

"Q"

Keyboard

n-bit system bus

n

Copyright © 2004 Pearson Prentice Hall, Inc.

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   The SPARC Processor State
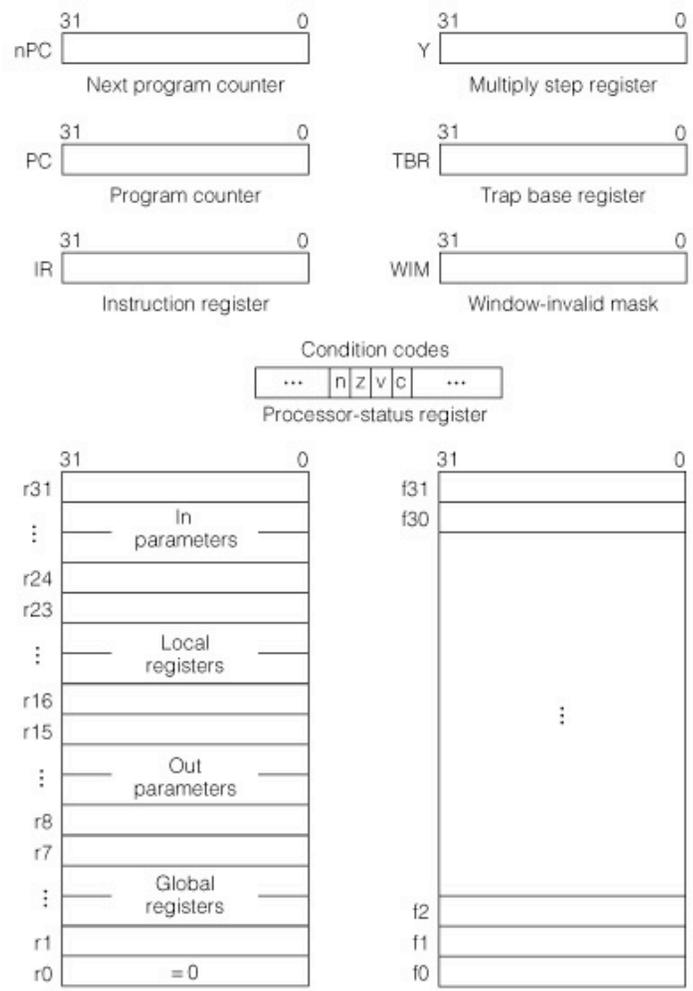
Fig. 3.10 Register Windows: an Important Concept in SPARC
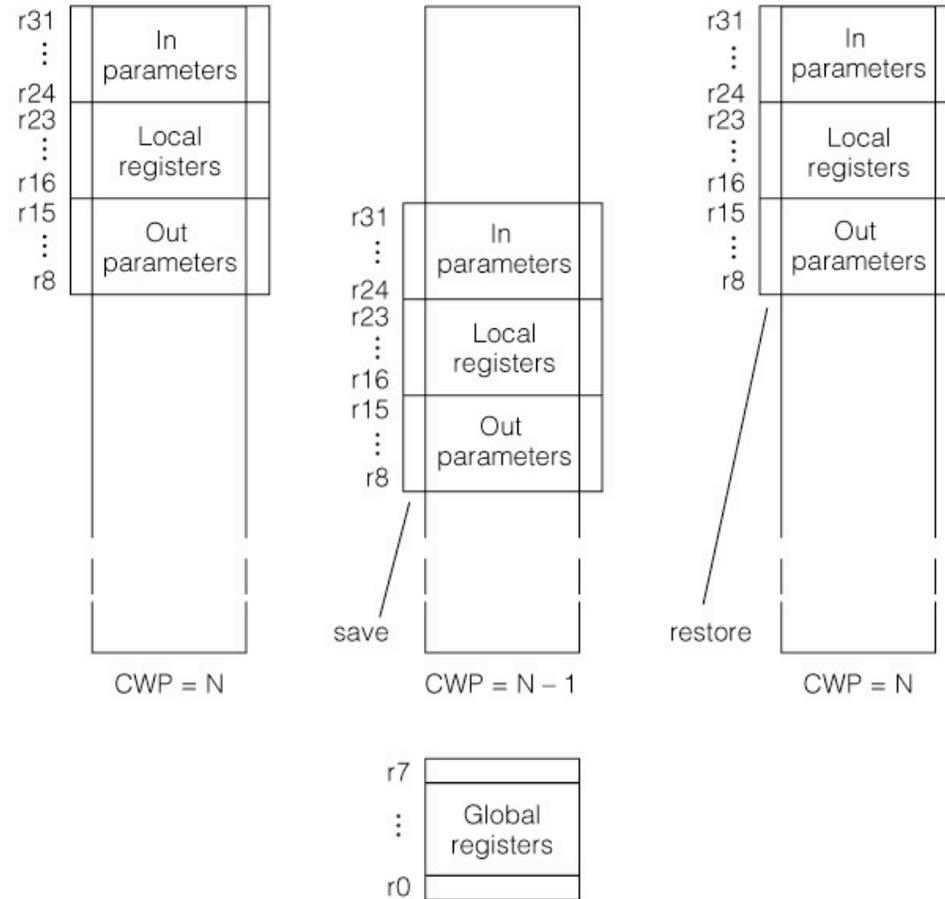
Copyright © 2004 Pearson Prentice Hall, Inc.

SPARC Memory

RTN for the SPARC memory:

$Mb[0..2^{32}-1]\langle 7..0\rangle$:   Byte memory;

$Mh[a]\langle 15..0\rangle := Mb[a]\langle 7..0\rangle\#Mb[a+1]\langle 7..0\rangle$:   Halfword memory;

$M[a]\langle 31..0\rangle := Mh[a]\langle 15..0\rangle\#Mh[a+2]\langle 15..0\rangle$:   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
- Regs. 8-15 are used for incoming parameters
- Regs. 24-31 are for outgoing parameters
- Current Window Pointer CWP locates reg. 8
- Overflow of reg. 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 & 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 reg., index in another

    - Register indirect: G0+Gn

    - Displacement: Gn+const, n≠0

    - Absolute: G0+const.

- Absolute addressing can only reach the bottom or top 4K bytes of memory

RTN for SPARC Instruction Formats

$op\langle 1..0\rangle := IR\langle 31..30\rangle$:   Instruction class, op code for format 1;

$disp30\langle 29..0\rangle := IR\langle 29..0\rangle$:   Word displacement for call, format 1;

$a := IR\langle 29\rangle$:   Annul bit for branches, format 2a;

$cond\langle 3..0\rangle := IR\langle 28..25\rangle$:   Branch condition select, format 2a;

$rd\langle 4..0\rangle := IR\langle 29..25\rangle$:   Destination register for formats 2b & 3;

$op2\langle 2..0\rangle := IR\langle 24..22\rangle$:   Op code for format 2;

$disp22\langle 21..0\rangle := IR\langle 21..0\rangle$:   Constant for branch displacement or sethi;

$op3\langle 5..0\rangle := IR\langle 24..19\rangle$:   Op code for format 3;

$rs1\langle 4..0\rangle := IR\langle 18..14\rangle$:   Source register 1 for format 3;

$opf\langle 8..0\rangle := IR\langle 13..5\rangle$:   Sub-op code for floating point, format 3a;

$i := IR\langle 13\rangle$:   Immediate operand indicator, formats 3b & c;

$simm13\langle 12..0\rangle := IR\langle 12..0\rangle$:   Signed immediate operand for format 3c;

$rs2\langle 4..0\rangle := IR\langle 4..0\rangle$:   Source register 2 for format 3b.

Fig. 3.11   SPARC Instruction Formats

Format number

SPARC instruction formats

1. Call

31 30 29	0
0 1	disp30

2a. Branches

31 30 29 28	25 24	22 21	0
0 0	a	cond	op2	disp22

2b. sethi

| 0 0 | rd | op2 | disp22 |

3a. Floating point

31 30 29	25 24	19 18	14 13 12	5 4	0
op	rd	op3	rs1	opf	rs2

3b. Data movement

| op | rd | op3 | rs1 | 0 | asi | rs2 |

3c. ALU

| op | rd | op3 | rs1 | 1 | simm13 |

i (register or immediate)

- Three basic formats with variations

$adr\langle 31..0\rangle := (i=0 \rightarrow r[rs1] + r[rs2]$:   Address for load, store,
                    $i=1 \rightarrow r[rs1] + simm13\langle 12..0\rangle$ {sign ext.}):   and jump;
$calladr\langle 31..0\rangle := PC\langle 31..0\rangle + disp30\langle 29..0\rangle$ #00$_2$:   Call relative address;
$bradr\langle 31..0\rangle := PC\langle 31..0\rangle + disp22\langle 21..0\rangle$ #00$_2$ {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
ar	10	00 0010	Rdst ← Rsrc1 OR Rsrc2 (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 #upper22, R17

    OR R17, #lower10, R17

- Double words 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 Typical SPARC Arithmetic Instructions

Inst.	OPCODE	Meaning
add	0X 0000	Add or add and set condition codes
addc	0X 1000	Add with carry: set CCs or not
sub	0X 0100	Subtract: set CCs or not
subc	0X 1100	Subtract with borrow: set CCs or not
mulscc	10 1100	Do one step of multiply

- All are format 3, Op=10

- CCs are set if X=1 and not if X=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.	OPCODE	Meaning
AND	0S 0001	AND, set CCs if S=1 or not if S=0
ANDN	0S 0101	NAND, set CCs or not
OR	0S 0010	OR, set CCs or not
ORN	0S 0110	NOR, set CCs or not
XOR	0S 0011	XNOR(Equiv), set CCs or not
SLL	10 0101	Shift left logical, count in RSRC2 or imm13
SRL	10 0110	Shift right logical, count in RSRC2 or imm13
SRA	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 & undisturbed if S=0

Tbl. 3.11 SPARC Branch and Control Instructions

Inst.	Fmt.	Op	OPCODE or Op2	Meaning
ba	2	00	010	Unconditional branch
bcc	2	00	010	Conditional branch
call	1	01		Call & save PC in R15
jmpl	3		11 1000	Jmp to EA, save PC in Rdst
save	3		11 1100	New register window, & ADD
restore	3		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 Code: add two integers

```
        .begin
        .org
progl:  ldw     [x],      %r1          ! load a word from M[x] into register %r1
        ldw     [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
```

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
```
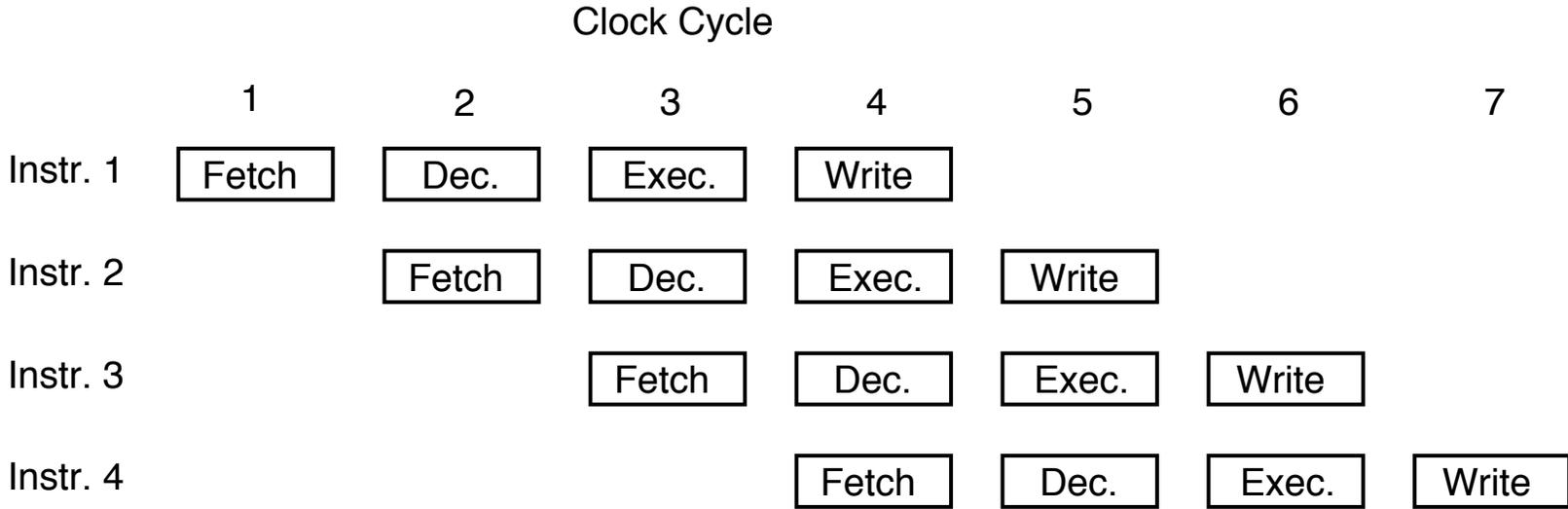
Pipelining of the SPARC Architecture

- 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 inst. while executing current inst.
- Pipelining breaks inst. processing into steps
  - A step of one instruction overlaps different steps for others
- A new inst. is started (issued) before previously issued instructions are complete
- Instructions guaranteed to complete in order

Fig. 3.14 The SPARC MB86900 Pipeline

Clock Cycle

	1	2	3	4	5	6	7
Instr. 1	Fetch	Dec.	Exec.	Write			
Instr. 2		Fetch	Dec.	Exec.	Write		
Instr. 3			Fetch	Dec.	Exec.	Write	
Instr. 4				Fetch	Dec.	Exec.	Write

- 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 inst. 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 vs. 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 vs. CISC

C S D A 2/e

- While a RISC machine may possibly have fewer instructions than a CISC, the instructions are always simpler. Multi-step 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 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.