CPE300: Digital System Architecture and Design

Fall 2011
MW 17:30-18:45 CBC C316

CISC: Motorola MC68000
09262011

http://www.egr.unlv.edu/~b1morris/cpe300/

Outline

- Recap
- CISC vs. RISC
- Motorola MC68000

Machine Representation

- Computers manipulate bits
  - Bits must represent “things”
    - Instructions, numbers, characters, etc.
    - Must tell machine what the bits mean
- Given N bits
  - 2^N different things can be represented

Positional Notation for Numbers

- Base (radix) B number →B symbols per digit
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6 , 7, 8, 9
  - Base 2 (binary) 0, 1
- Number representation
  - $d_{31}d_{30}...d_2d_1d_0$ is 32 digit number
  - Value = $d_{31}\times B^{31} + d_{30}\times B^{30} + \ldots + d_1\times B^1 + d_0\times B^0$

Two's Complement

- Unbalanced representation
  - Leading zeros for positive
    - 2^{N-1} non-negatives
  - Leading ones for negative number
    - 2^{N-1} negative number
  - One zero representation
- First bit is sign-bit (must indicate width)
  - Value = $d_{31} \times -2^{31} + d_{30} \times 2^{30} + \ldots + d_1 \times 2^1 + d_0 \times 2^0$

Negative value for sign bit

Two's Complement Notes

- Negation shortcut
  - Invert bits and add 1
    - $\bar{x} + 1 = -x$
- Sign extension
  - Replicate sign bit (msb) of smaller container to fill new bits in larger container
- Overflow
  - Not enough bits to represent a number
  - Indicated by V flag in condition code

Machine Performance

- What is machine performance?
- How can performance be measured?

- Response time
  - How long to complete a task
- Throughput
  - Total work completed per unit time
  - E.g. task/per hour

- Program execution time is best measure of performance

Relative Performance

- $\text{Performance}_x = \dfrac{1}{\text{Execution time}_x}$
- $\text{Speedup} = n = \dfrac{\text{Performance}_x}{\text{Performance}_y} = \dfrac{\text{Execution time}_y}{\text{Execution time}_x}$

Performance Summary

- CPU Time = #Instructions × CPI × clock cycle time

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

- T := CPU time
- IC := instruction count
  - Number of instruction in a program
- CPI := clock cycles/instruction
  - Average clock cycles per instruction – depends on instruction → calculate by instruction mix
- τ := duration of clock period
  - Specified in time or in rate (Hz)

RISC vs. CISC 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

Memory Bottleneck

- Memory no longer expensive
- Design for speed

Parameter	1981 (8086)	2004 (Pentium P4)	Improvement factor
Clock Frequency	4.7 MHz	4 GHz	~1000
Clock Period	212 ns	200 ps	~1000
Memory Cycle Time	100 ns	70 ns	1.4
Clocks per Memory Cycle	.47	280	~ -500

Dealing with Memory Bottleneck

- Employ one or more levels of cache memory.
  - Prefetch instructions and data into I-cache and D-cache.
  - Out of order execution.
  - Speculative execution.
- One word per instruction (RISC)
- Simple addressing modes (RISC)
- Load-Store architecture (RISC)
- Lots of general purpose registers (RISC)

RISC Design Characteristics

- 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

More on RISC

- Prefetch instructions
  - Get instruction/data/location before needed in pipeline
- Pipelining
  - Beginning execution of an instruction before the previous instruction(s) have completed. (Chapter 5.)
- Superscalar operation
  - Issuing more than one instruction simultaneously.
  - Instruction-level parallelism (Chapter 5.)
- Out-of-order execution
- 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 (Chapter 3)

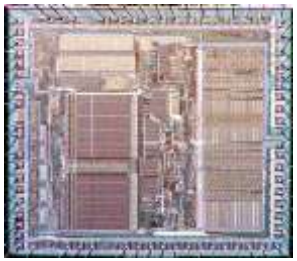
Developing and ISA (Table 3.1)

- Memories: structure of data storage in the computer
  - Processor-state registers
  - Main memory organization
- Formats and interpretation: meaning of register fields
  - Data types
  - Instruction format
  - Instruction address interpretation
- Instruction interpretation: things done for all instructions
  - Fetch-execute cycle
  - Exception handing
- Instruction execution: behavior of individual instructions
  - Grouping of instructions into classes
  - Actions performed by individual instructions

The Motorola MC6800

chip

- Introduced in 1979
  - Computers
    - Apple Lisa 2, Apple Macintosh 128, Atari 520STfm and 1040STfm, Commodore Amiga 500 and 1000
  - Still in use today (now Freescale Semiconductor)
- Very early 32-bit microprocessor
  - Most operations on 32-bit internal data
  - Some operations may use different number of bits
  - External datapaths may not all be 32 bits wide
    - 24-bit address bus for MC68000
- Complex instruction set computer
  - Large instruction set
  - 14 addressing modes

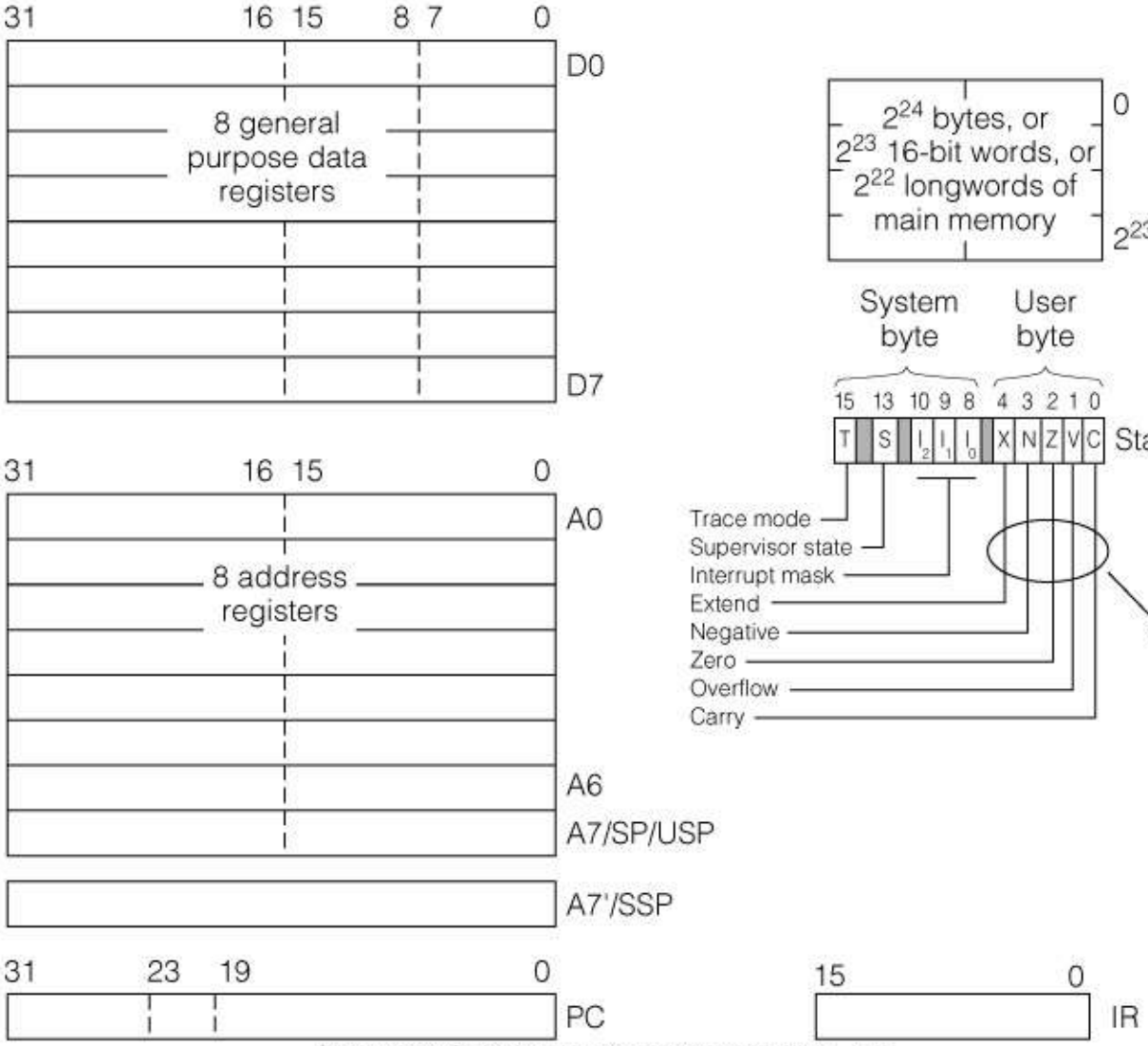


die

New Concepts from MC68000

- Effective address (EA)
  - Addressing modes
- Subroutines
  - E.g. function calls
- Starting a program
  - Assemble, link, load, and run times
- Exceptions
  - Interruption of normal sequential instruction execution
- Memory-mapped I/O
  - Part of CPU memory reserved for I/O

MC68000 Programmer's Model



Features of 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
    - 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 and the Interrupt Mask

RTN Processor State

- Registers
  - `D[0..7]<31..0>:` General purpose data registers;
  - `A[0..7]<31..0>:` Address registers;
- `PC<23..0>:` Program counter (original MC68000)
- `IR<15..0>:` Instruction register;
- Stack pointers
  - `SP := A[7]:` User stack pointer, also called USP;
  - `A7´<31..0>:` System stack pointer;
  - `SSP := A7´:` System stack pointer;
- `Status<15..0>:` System status byte and user status byte;
- User byte (condition codes)
  - `C := Status<0>:` Carry flag
  - `V := Status<1>:` oVerflow flag;
  - `Z := Status<2>:` Zero flag
  - `N := Status<3>:` Negative flags;
  - `X := Status<4>:` Extend flag;
- System byte
  - `INT<2..0> := Status<10..8>:` Interrupt mask;
  - `S := Status<13>:` Supervisor state flag;
  - `T := Status<15>:` Trace mode flag;

Main Memory

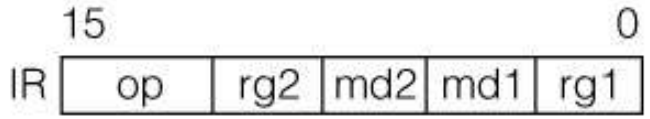
- Main memory:
  - `Mb[0..2^24-1]<7..0>:` memory as bytes
  - `Mw[ad]<15..0> := Mb[ad]#Mb[ad+1]:` memory as words
  - `Ml[ad]<31..0> := Mw[ad]#Mw[ad+2]:` memory as longwords
- Word and longword forms are big-endian
  - Lowest numbered byte contains most significant bit of word
- Hard word alignment constraints
  - Not described in the RTN
  - Word addresses must in end in on binary 0
  - Longword addresses end in two binary 0
    - What are differences between soft alignment?

Operand Types

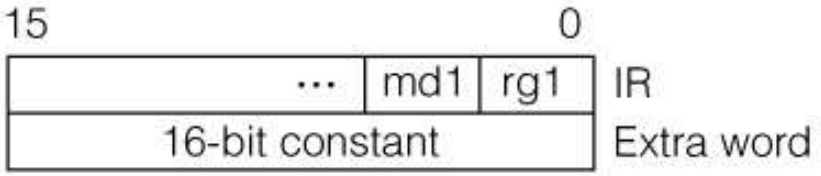
- One instruction may operate on several types (CISC design)
  - `MOVE.B` bytes
  - `MOVE.W` word
  - `MOVE.L` longwords
  - Default is word operands
  - Operand length encoded in instruction
- Bits to encode operand type vary with instruction
  - Assumption for RTN description
    - `d :=datalen(IR):`
  - Function returns 1, 2, 4 for operand length

Instruction Formats

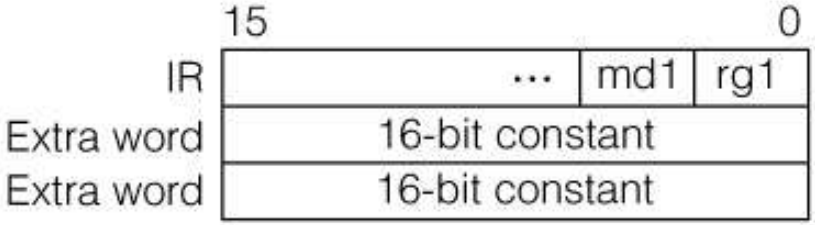
- Instructions accessed in 16-bit words
- Variable number of words in an instruction



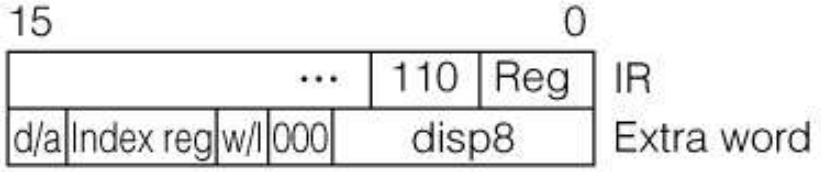
(a) A 1-word move instruction



(b) A 2-word instruction



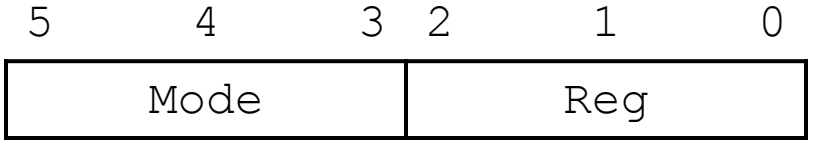
(c) A 3-word instruction



(d) Instruction with indexed address

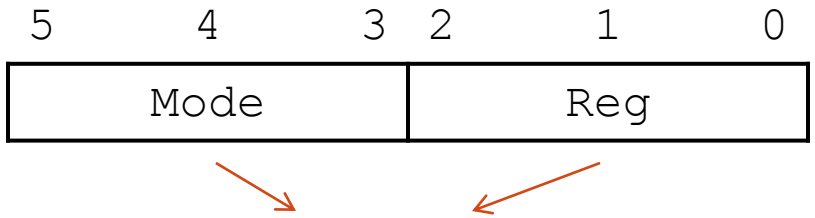
Addressing Modes

- General address of operand specified by 6-bit field
  - Access paths to memory and registers
  - See Table 3.2 for details
- 6-bit effective address



  - Mode field provides access paths to operands
- Not all operands/results can be specified by general address - some must be in registers
- Exception
  - Destination of MOVE instruction has mode and reg fields reversed

Table 3.2: MC68000 Addressing Modes



Name	Mode	Reg	Assembler Syntax	Extra Words	Description
Data register direct	0	0-7	Dn	0	Dn
Address register direct	1	0-7	An	0	An
Address register 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 indexec 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]
Relative indexed short	7	3	disp8(PC,XnLo)	1	M[PC+XnLo+disp8]
Relative indexed long	7	3	disp8(PC,Xn)	1	M[PC+Xn+disp8]
Immediate	7	4	#data	1-2	No location, data

RTN Description of Addressing

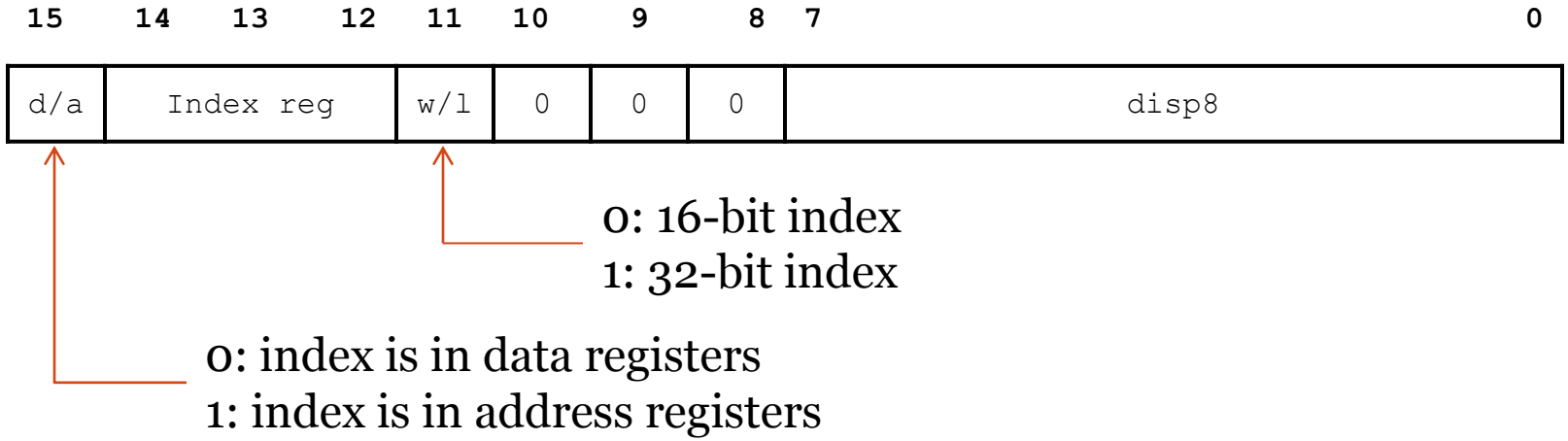
- Addressing modes interpret many items
  - Instruction in the IR register
  - Following 16-bit word: `Mw[PC]`
  - D and A registers in CPU
- Many addressing modes calculate an effective memory address
- Some modes designate a register
- Some modes result in constant operand
- Restrictions exist for some modes

RTN Formatting for EA Calculation

- `XR[0..15]<31..0> := D[0..7]<31..0>#A[0..7]<31..0>:`
- `xr<3..0> := Mw[PC]<15..12>:`
- `wl := Mw[PC]<11>:`
- `dsp8<7..0> := Mw[PC]<7..0>:`
- `index := ((wl=0) → XR[xr]<15..0>: (wl=1) → XR[xr]<31..0>):`

- Index register can be D or A
- Index number for index mode
- Short /long index flag
- Displacement for index mode
- Short
- Long index value

- 4-bit field specifies index register
- Either 16 or 32 bit s of index register may be used
- Low order 8-bits are used as offset



Calculating EA

- `md` and `rg` are 3-bit mode and reg fields
- `ea` is effective address

- Define effective address based on mode and register fields
  - `ea(md, rg) := (`
    - `(md=2) → A[rg<2..0>]`
    - `(md=3) → (A[rg]; A[rg] ← A[rg] + d):`
    - …

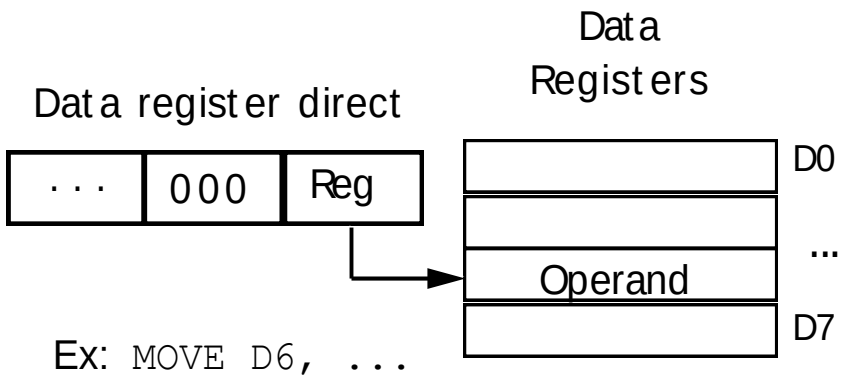
Addressing Mode Highlights

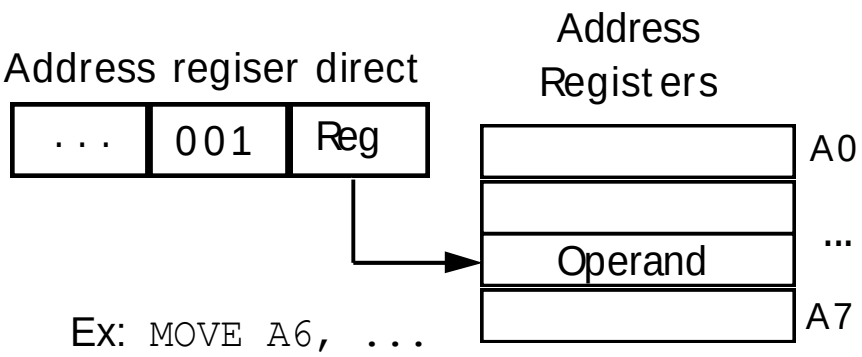
- Modes 0-6 use a register to calculate a memory address
  - Based modes (5-6) require an extra word (16-bits) to specify address
- Mode 7 does not use a register
  - Functionality is expanded by repurposing `reg` field
  - All variants require extra words to complete the instruction and specify the memory address

Mode 0 and 1: Register Direct

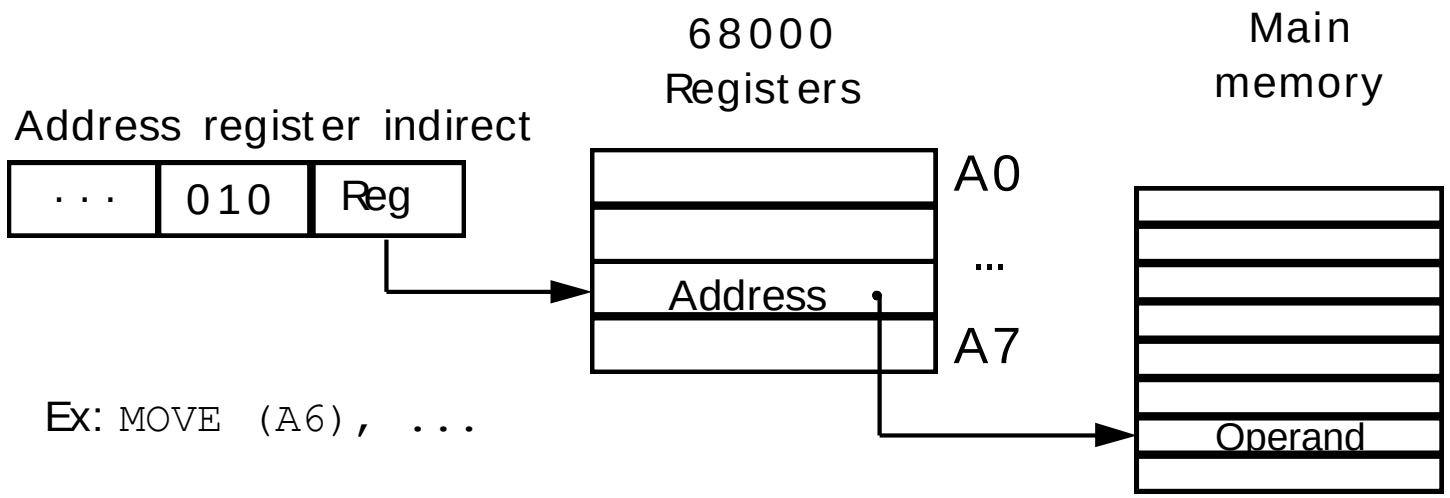
- Mode 1 = data register
- Mode 2 = address register

- Register itself provides place to store result or location of operand
  - No memory address in this mode



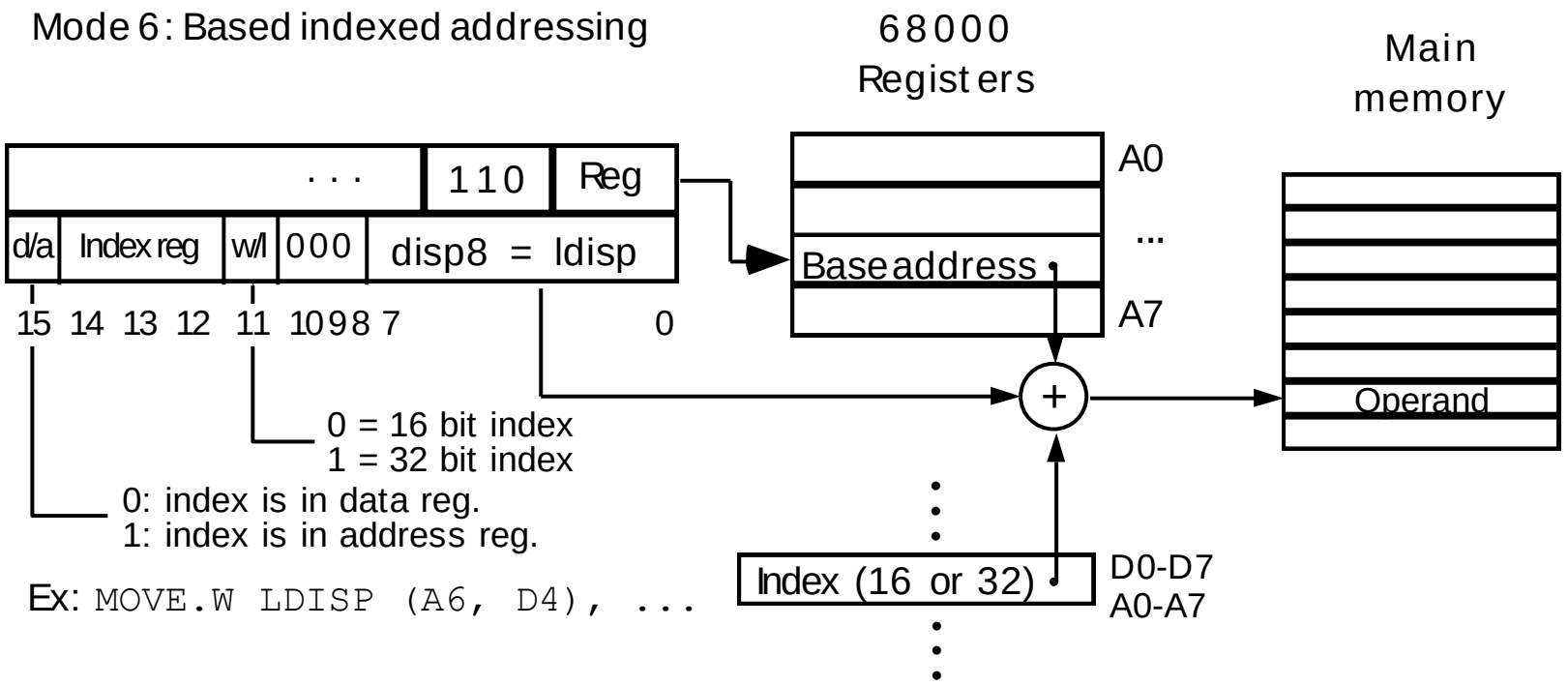


Mode 2: Address Register Indirect



- Modes 3 and 4 are the same
  - Autoincrement (3) – register incremented after obtained
  - Autodecrement (4) – register decremented before address obtained

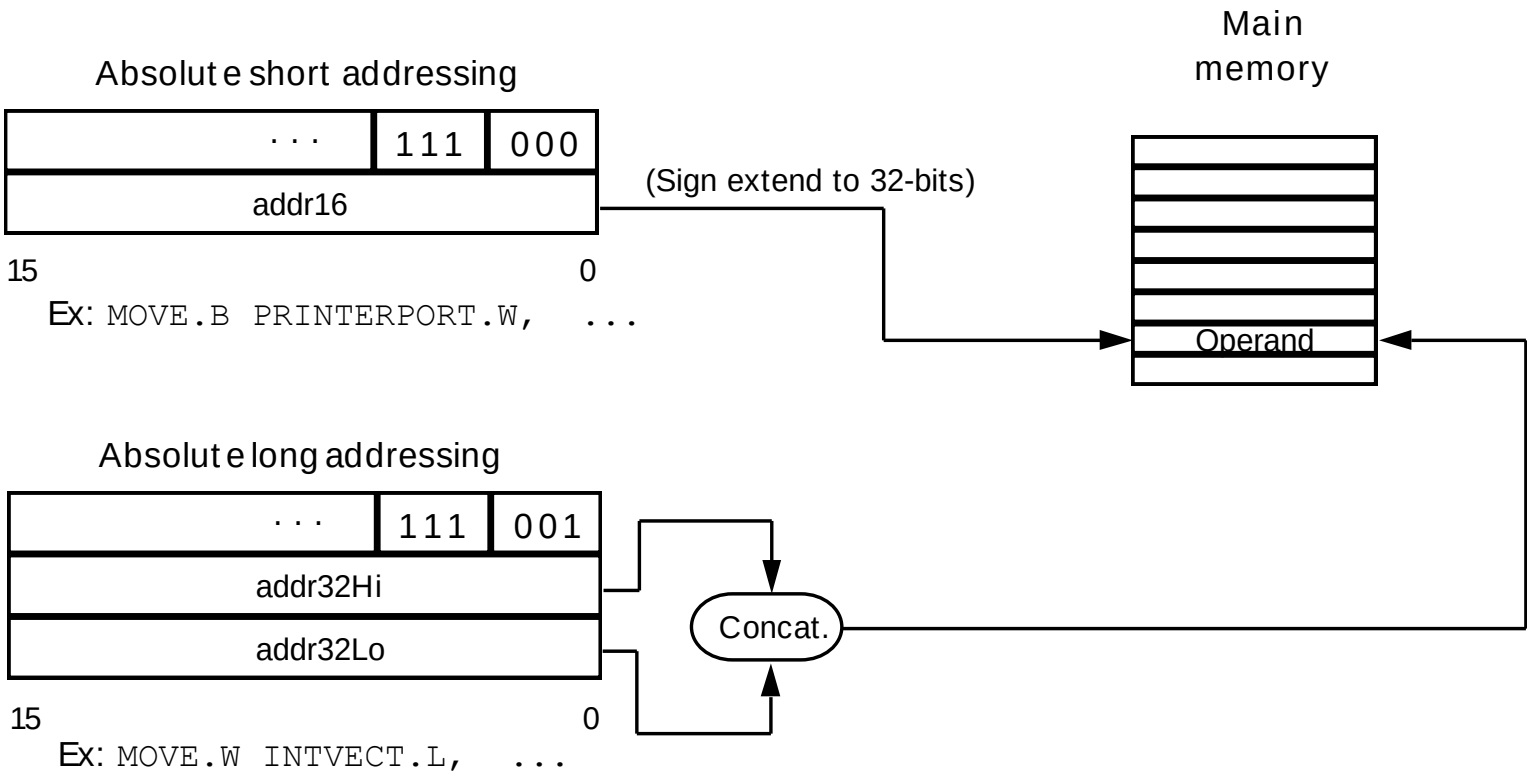
Mode 6: Based Indexed



- Three items added to get address
- Mode 5 (based) is same only does not contain register index

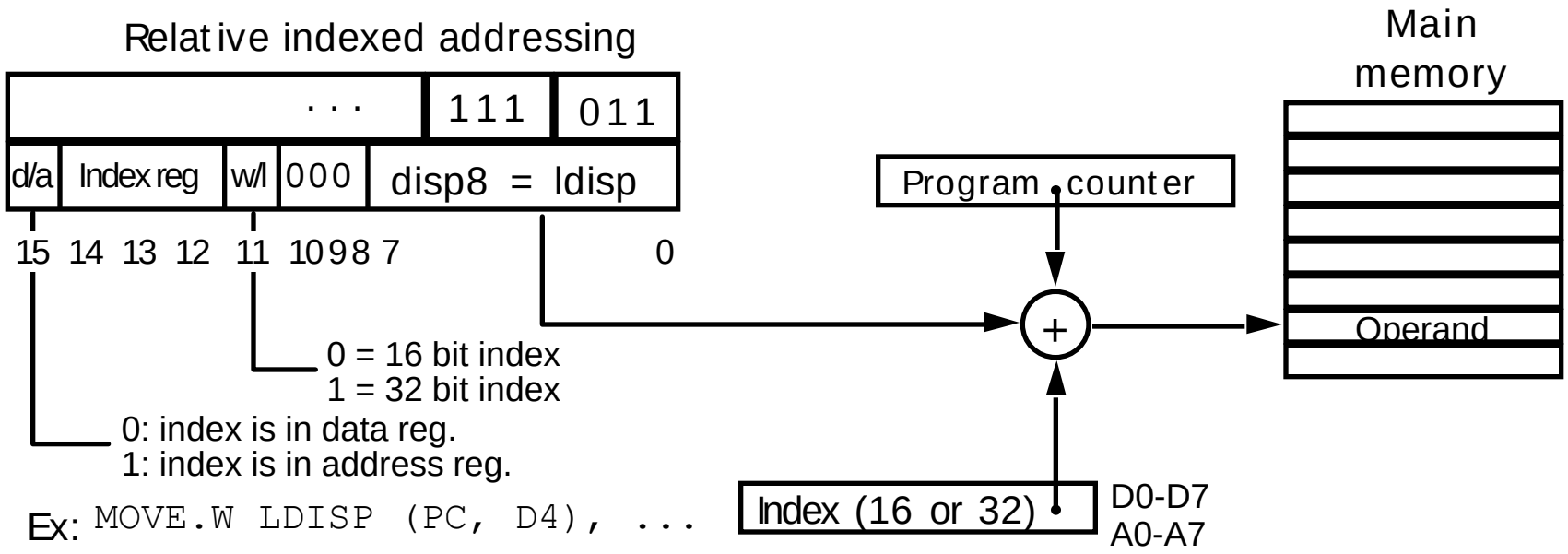
Mode 7-0 and 7-1: Absolute Addressing

- Mode 7-0 – 16-bit addresses
- Mode 7-1 – 32-bit addresses



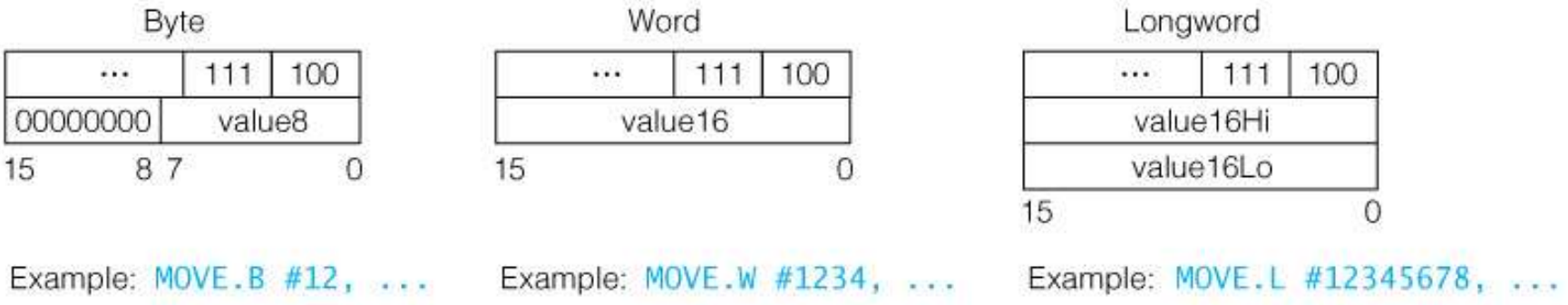
Mode 7-3: Relative Indexed

- Same as indexed mode but uses PC rather than an A register as base address



Mode 7-4: Immediate

- Access to constants stored as part of the program
  - Constant stored immediately after the instruction word
  - Data length specified by oopcode field, not the `md/reg` field



Remember big endianess

Result Operand Addressing

- Not all addressing modes can be used for results
  - `md=7` and `rg=2` or `3` not allowed
  - Lead to self-modifying code

  - Register immediate is legal for results

Instruction Interpretation

- Instructions fetched 16-bits at a time
  - PC advanced by 2 as 16-bit word is fetched
  - Addressing mode may cause advance of 2 or 4 more words

- ```
  Instruction_interpretation := (
        Run → ( (IR<15..0> ← Mw[PC]<15..0>:
                  PC ← PC + 2);
        instruction_execution );  ):
  ```
  ```

Data Movement Instructions

- Unlike SRC, instruction fields are not standardized
  - Locations and sizes depend on instruction
  - Allows more instructions to be defined in small word size
- Condition codes can be set during move
  - Negative and zero

- `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
    ):
```

Integer ALU Instructions

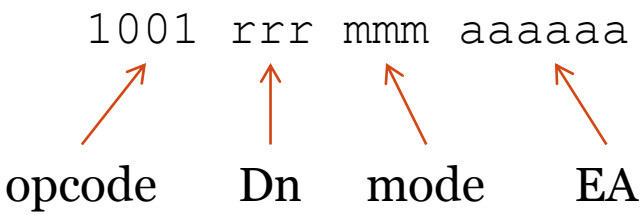
- 2-operand instructions must specify destination
  - One operand is EA
  - Second operand is Dn (data register direct)
  - 3-bit mode field specifies destination as EA or Dn and operands as byes, word, or long

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

Example: Subtract

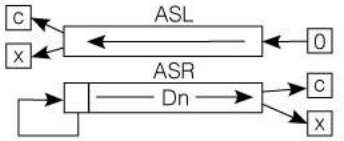
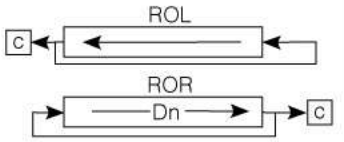
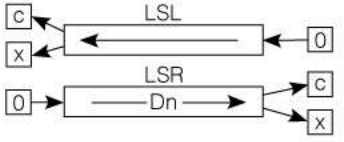
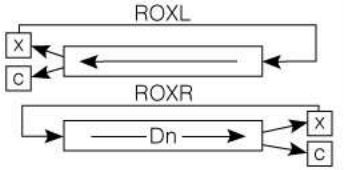
```
sub (:= op=9) → (
        (md2<2>=0) → D[rg2] ← D[rg2] - opnd(md1, rg1):
        (md2>2>=1) → (memval(md1, rg1) →
                                    (tmp ← ea(md1, rg1);
                                    M[tmp] ← M[tmp] - D[rg2]):
                    ¬memval(md1, rg1) →
                      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

- SUB EA, Dn



Arithmetic Shift and Rotates

- ww is word size
- Condition codes
  - N = msb of result
  - Z = set by result
  - C = last bit shifted out

Mnemonic	Operands	Opcode Word	XV	Operation
ASd	EA	1110000d11aaaaaa	xx	
ASd	#cnt,Dn	1110cccdww000rrr	xx	
ASd	Dm,Dn	1110RRRdww100rrr	xx	
ROd	EA	1110011d11aaaaaa	-0	
ROd	#cnt,Dn	1110cccdww011rrr	-0	
ROd	Dm,Dn	1110RRRdww111rrr	-0	
LSd	EA	1110001d11aaaaaa	x0	
LSd	#cnt,Dn	1110cccdww001rrr	x0	
LSd	Dm,Dn	1110RRRdww101rrr	x0	
ROXd	EA	1110010d11aaaaaa	x0	
ROXd	#cnt,Dn	1110cccdww010rrr	x0	
ROXd	Dm,Dn	1110RRRdww110rrr	x0	

Program Control Instructions

- Conditional branches
  - Use condition code bits (C, N, V, Z)
  - E.g. `BVS` = branch if overflow set
- `Bcc` = branch
- `DBcc` = decrement and branch
- `Scc` = sets result byte to outcome of test

Name	Meaning	Code	Logic	Name	Meaning	Code	Logic
T	True	0000	1	F	False	0001	0
CC	Carry clear	0100	$\bar{C}$	LS	Low or same	0011	$C + Z$
CS	Carry set	0101	C	LT	Less than	1101	$N \cdot \bar{V} + \bar{N} \cdot V$
EQ	Equal	0111	Z	MI	Minus	1011	N
GE	Greater or equal	1100	$\bar{N} \cdot \bar{V} + N \cdot V$	NE	Not equal	0110	Z
GT	Greater than	11100	$\overline{N \cdot V \cdot Z} + N \cdot V \cdot \bar{Z}$	PL	Plus	1010	$\bar{N}$
HI	High	0100	$\overline{C \cdot Z}$	VC	Overflow clear	1000	$\bar{V}$
LE	Less or equal	1111	$N \cdot \bar{V} + \bar{N} \cdot V + Z$	VS	Overflow set	1001	V

More Program Control

- Unconditional branches
  - Map to C goto statement
    - `BRA, JMP`
  - Sub-routine varieties store PC on stack
    - `BSR, JSR`
- Sub-routine return instructions
  - Linage uses stack for return address
  - `RTR, RTS`

Starting a Program

- Assembler
  - Convert assembly language text to (binary) machine language
    - Addresses translated using a symbol table
    - Addresses adjusted to allow room for blocks of reserved memory (e.g. an array definition)
- Linker
  - Separately assembled modules combined and absolute addresses assigned
- Loader
  - Move binary words into memory
- Run time
  - PC set to started address of loaded module.
  - OS usually makes a jump or procedure call to the address

Pseudo Operations

- Operation performed by assembler at assembly time not by CPU at run time
- `EQU` - defines constant symbol
  - `PI:  EQU  3.14`
  - Substitution made at assemble time
- `DS.(B, W, L)` – defines block of storage
  - A label is associated with first word of block
  - `Line:  DS.B  132`
  - Program loader (part of OS) accomplishes this
- `#` indicates value of symbol rather than location addressed by symbol
  - `MOVE.L  #1000, D0` ; moves 1000 to D0
  - `MOVE.L  1000, D0` ; moves value addr. 1000 to D0
  - Assembler detects difference
- `ORG` – defines memory address where following code will be stored
  - `Start:  ORG  $4000` ; next instruction/data at addr. 0x4000
- Character constants in single quotes
  - `'X'`

Example: Clearing Block of Memory

```
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 stack pointer
  - `A7` cannot be used for anything else

Exceptions

- Changes sequential instruction execution
  - Next instruction fetch not from PC location
  - Exception vector
    - Address supplying the next instruction
- Arise from instruction execution, hardware faults, external conditions
  - Interrupts – externally generated exceptions
  - ALU overflow, power failure, completion of I/O operation, out of range memory access, etc.
- Trace bit = 1 causes exception after every instruction
  - Used for debugging

Exception Handling Steps

1. Status change
   - Temporary copy of status register made
   - Supervisor mode bit S is set and trace bit T is reset
2. Exception vector address obtained
   - Small address made by shift 8-bit vector number left 2
   - Contents of longword at vector address is new address of next instruction
   - Exception handler or interrupt service routine starts at this address
3. Old PC and Status register are pushed onto supervisor stack, `A7' = SSP`
4. PC loaded from exception vector address
5. Return from handler is done by `RTE`
   - Works like `RTR` except Status register is restored rather than CCs

Exception Priority

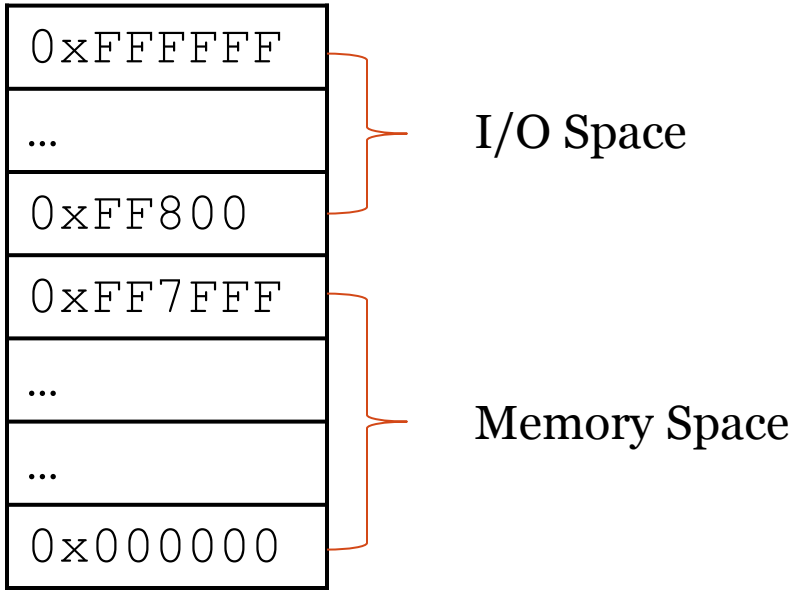
- Method to determine which exception vector to use when multiple exceptions occur at once
- 7 levels of priority in MC68000
  - Status Register contains current priority
- Exceptions with priority ≤ current priority are ignored

- Exceptions are sensed before fetching next instruction

Memory-Mapped I/O

- Part of CPU memory is devoted/reserved for I/O
  - No separate I/O space
  - Not popular for machines having limited address bits
- Single bus needed for memory and I/O
  - Less packaging pins
- Size 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 the top end of address space

24-bit address space with top 32K reserved for I/O



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