CIS2030 Microcomputers Week 1

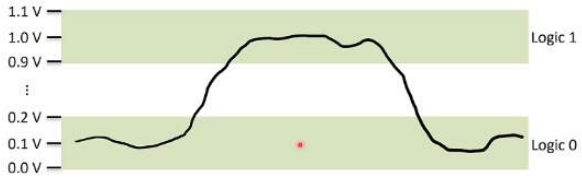
Binary Encodings

Number Systems

- Binary (base 2) - hardware
  - $2\ symbols\ \{0, 1\}$
- Octal (base 8) - software
  - $8\ symbols\ \{0, 1, 2, 3, 4, 5, 6, 7\}$
  - Power of 2: $2^3 = 8$ (each value represented by 3 bits)
- Decimal (base 10) - software
  - $10\ symbols\ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Hexadecimal (base 16) - software
  - $16\ symbols\ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
  - Power of 2: $2^4 = 16$ (each value represented by 4 bits)

Digital Signals and Binary

- All information stored and processed by the computer is encoded as strings of bits
- Why?
  - Bits can be stored in and retrieved from bi-stable devices
  - Bits can be reliably transmitted on noisy inaccurate wires
- Analog Computing



To Decimal Conversion

$(1234)_{10} \rightarrow ?$

$4 \times 10^0 = 4$
$3 \times 10^1 = 30$
$2 \times 10^2 = 200$
$1 \times 10^3 = 1000$
$+1234$

$(1011)_2 \rightarrow ?_{10}$

$1 \times 2^0 = 1$
$1 \times 2^1 = 2$
$0 \times 2^2 = 0$
$1 \times 2^3 = 8$
$+ 11_{10}$

$(106)_8 \rightarrow ?_{10}$

$6 \times 8^0 = 6$
$0 \times 8^1 = 0$
$1 \times 8^2 = 64$
$+ 70_{10}$

$(104)_{16} \rightarrow ?_{10}$

$4 \times 16^0 = 4$
$0 \times 16^1 = 0$
$1 \times 16^2 = 256$
260_{10}

Successive Division

- Any number D in base 10 is equivalent to an n-digit number in base R

$$D = (a_{n-1}a_{n-2}\ldots a_1a_0)_R$$

$$= a_{n-1}R^{n-1} + a_{n-2}R^{n-2} + \ldots + a_1R^1 + a_0R^0$$

- As we divide by R, each subsequent division liberates the next lower-order digit a_i from the bas R representation

$Q_0 = D/R = a_{n-1}R^{n-2} + a_{n-2}R^{n-3} + \ldots + a_1R^0$ remainder a_0

- So the first division yields the quotient Q_0, and the remainder a_0 - the value of the lowest order digit

$Q_1 = Q_0/R = a_{n-1}R^{n-3} + a_{n-2}R^{n-4} + \ldots + a_2R^0$ remainder a_1 ...

$Q_{n-1} = Q_{n-2}/R = 0$ remainder a_{n-1}

- If we repeat the process n times, we eventually obtain a quotient of 0 and remainder of a_{n-1}, the high highest order digit in base R
- An example of converting decimal to binary:

$45_{10} \rightarrow ?_2$

2 | 45		
2 | 22	R=1	(least-significant bit (lsb))
2 | 11	R=0	
2 | 5	R=1	
2 | 2	R=1	
2 | 1	R=0	
0	R=1	(most-significant bit (msb))

101101_2

Real-Number Conversion

$3.65_{10} \rightarrow ?_2$

successive division — successive multiplication

$R \times \quad . A_{-1} R^{-1} + A_{-2} R^{-2} + A_{-3} R^{-3} + \ldots$

$A_{-1} . A_{-2} R^{-1} + A_{-3} R^{-2} + \ldots$

11_2

0.65	x	2	=	1.30	1	(msb)
0.30	x	2	=	0.60	0	
0.60	x	2	=	1.20	1	
0.20	x	2	=	0.40	0	repeat
0.40	x	2	=	0.80	0	
0.80	x	2	=	1.60	1	
0.60					↑	

$11.10\overline{1001}$

- Some decimal fractions will need approximations in binary: m x 2^n
- Remember the zeroes when working to the right of the binary point

Binary Addition

- Rules:
  - Add numbers and discard final carry
  - If result is negative, its msb = 1 and it's in 2's complement form
  - Assume a 5-bit word unless said otherwise
- Very efficient in hardware

Rules:

0	0	1
+ 0	+ 1	+ 1
0	1	10

Assume a 5-bit word.

```
 12        0 1 1 0 0
+ 6      + 0 0 1 1 0
 18        1 0 0 1 0 ✓
           ↑     ↑
           16 + 2 = 18
```

Binary Subtraction

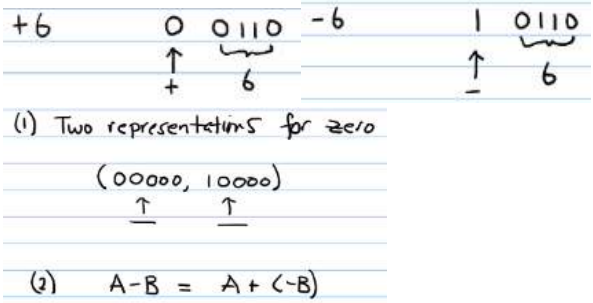
- Very inefficient in hardware

```
                0 ~~1~~ 10
 9            0 ~~1~~ 0 0 1
-3          - 0 0 0 1 1
 6            0 0 1 1 0
                ↑ ↑
               4+2 = 6
```

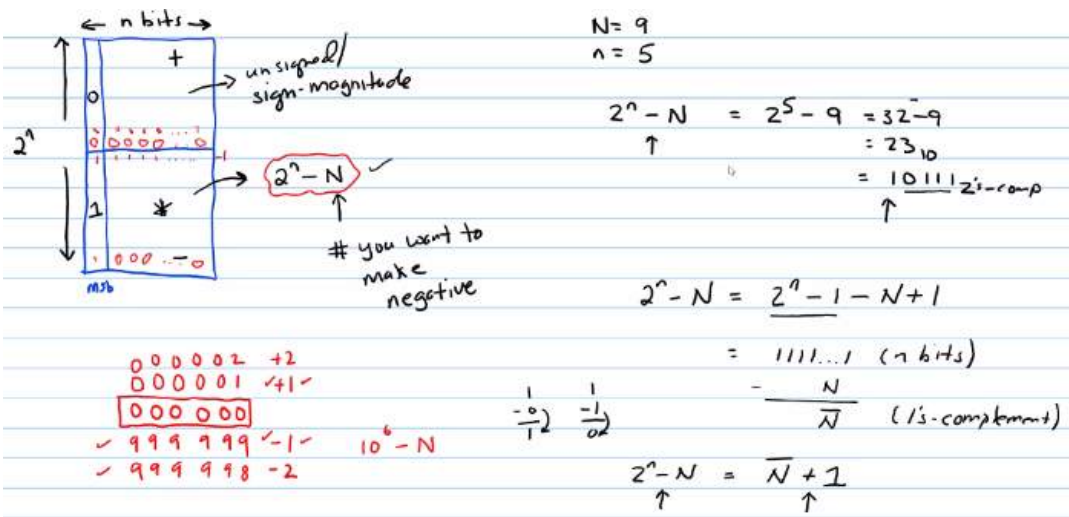
- A - B = A + (-B) is efficient

Signed Numbers

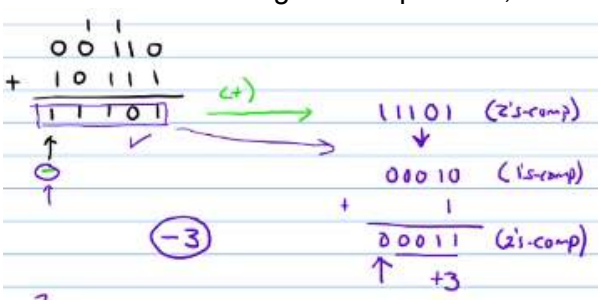
- Sign-magnitude



- 1's complement
  - Negate bits (0 -> 1 & 1 -> 0)
- 2's complement
  - 2's complement (N) = 1's complement(N) + 1 => $\underline{N}$+ 1
  - Assume an n-bit word
  - For -9:



- Machine doing 2's complement, adds ASCII (-) to front and converts number to positive



- Rules:
  - Leave all least-significant 0's as is
  - Leave first least-significant 1 as is

- Flip remaining bits

Range

k = 3 bits

k-bits	unsigned	sign-magnitude	2's-complement
0 0 0	0	+0	+0
0 0 1	1	+1	+1
0 1 0	2	+2	+2
0 1 1	3	+3	+3
1 0 0	4	-0	-4
1 0 1	5	-1	-3
1 1 0	6	-2	-2
1 1 1	7	-3	-1

unsigned: 0 to $2^k - 1$

sign-magnitude: $-(2^{k-1} - 1)$ to $+(2^{k-1} - 1)$

2's-complement: $-(2^{k-1})$ to $+(2^{k-1} - 1)$

Overflow

- Assume a word size of k = 6 bits

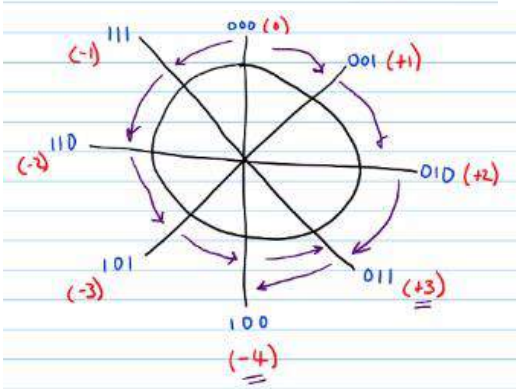
```
  30                    1 1 1 1
 +30                    0 1 1 1 1 1 0
 ---                  + 0 1 1 1 1 1 0
  60            (0)    -------------
  ↑             ↑      1 1 1 1 0 0
                Cout
```

* unsigned: 0 to $2^6 - 1$

0 to 63 ✓ ⇒ No overflow!

* 2's-comp: -32 to $+31$ × ⇒ overflow!

Unsigned: if (Cout == 0)
No overflow
else
Overflow



2's - complement:

$$* \quad V = \overline{A_{n-1}} \cdot \overline{B_{n-1}} \cdot R_{n-1} + A_{n-1} \cdot B_{n-1} \cdot \overline{R_{n-1}}$$

$$= \overline{0} \cdot \overline{0} \cdot 1 \qquad 1 \cdot 1 \cdot \overline{0}$$

$$= 1 \cdot 1 \cdot 1 \qquad 1 \cdot 1 \cdot 1$$

$$= 1 \ (T) \qquad 1 \ (T)$$

$$\begin{array}{r} \boxed{A_{n-1}}\ A_{n-2} \ \ldots \ A_0 \\ + \ \boxed{B_{n-1}}\ B_{n-1} \ \ldots \ B_0 \\ \hline \boxed{R_{n-1}}\ R_{n-2} \ \ldots \ R_0 \end{array} \qquad A_i, B_i, R_i \in \{0,1\}$$

sign

Signed, Unsigned, Word Size and Range

	Word Size in Bits			
	8	**16**	**32**	**64**
U_{max}	255	65,535	4,294,967,295	18,446,744,073,709,551,615
S_{max}	127	32,767	2,147,483,647	9,223,372,036,854,775,807
S_{min}	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- C Programming
  - Header file <limits.h> limits the values of various variable types
    - CHAR_BIT = 8
    - SCHAR_MIN = -128
    - SCHAR_MAX = 127
    - UCHAR_MAX = 255
    - ULONG_MAX = 18446744073709551615

C Programming - Constants, Integers and Casting

- Constants
  - Are signed integers by default
  - Add the suffix “U” to make unsigned: 123U
- Integers
  - Are signed by default: int a, unsigned b;
  - Explicit casting can be performed between signed and unsigned values
    - a = (int) b;
    - b = (unsigned) a;
  - Implicit casting occurs when performing assignments and function calls
    - a = b
    - int foo(unsigned u)
  - Example: K = 16 bits, U_{max} = 65535, S_{min} = -32768, S_{max} = 32767

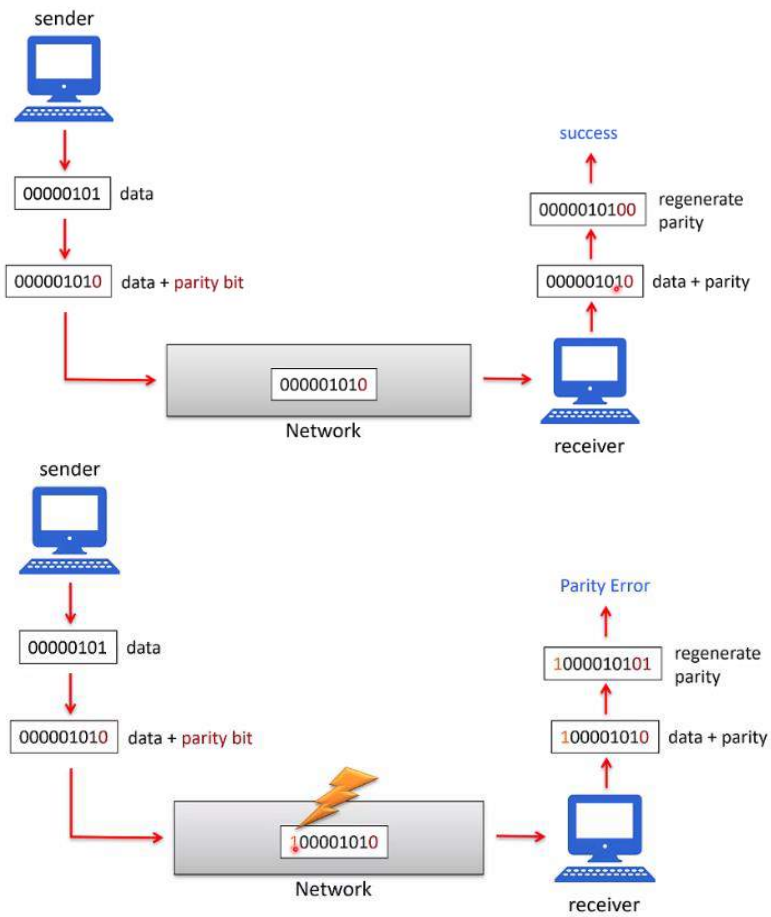
Constant1	Constant2	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
32767	-32768	>	signed
32767U	-32768	<	unsigned
-1	-2	>	signed
(unsigned) -1	-2	>	unsigned
32767	32768U	<	unsigned
32767	(int) 32768U	>	signed

Parity

- One of the most common ways to achieve error detection
    - Extra “parity” bit included with data to make total 1’s odd or even

Original Data	Even Parity	Odd Parity
000	0	1
001	1	0
010	1	0
011	0	1
100	1	0
101	0	1
110	0	1
111	1	0

- Example of using (even) parity:



ASCII

- 7-bit code
  - 33 control
  - 95 graphic

Bits 4321	Bit Positions 765							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Fixed-Floating Point Representations

Binary Fractions

- Decimal number $=\sum_{k=-j}^{i} b_k 2^k$

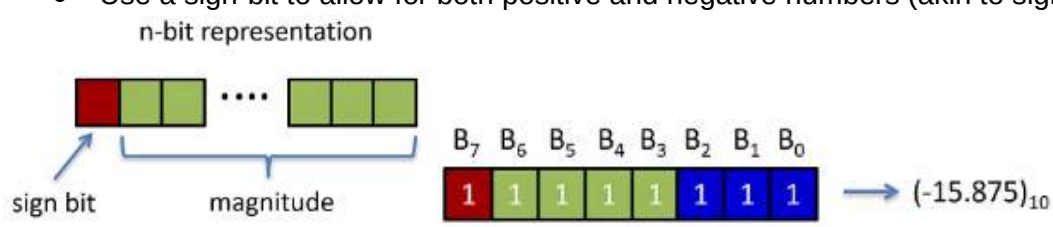
Representable Values

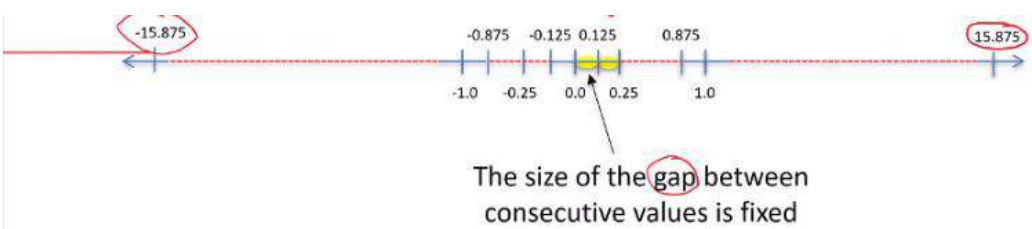
- Primary limitation of fractional binary numbers
  - Can only exactly represent numbers that can be written as m x 2^n
  - Other rational numbers have repeating bit representations

Decimal Fraction	Binary Representation
1/3	$0.\overline{01}$
1/5	$0.0011001100110\overline{0011}$
65/100	$0.10\overline{1001}$

Fixed-Point Representation

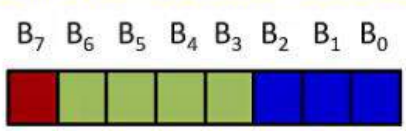
- Pick a fixed point for the implied binary port
- Use a sign-bit to allow for both positive and negative numbers (akin to sign-magnitude)





- The position of the binary point affects the range and precision of the representation
  - **Range**: difference between largest and smallest representable numbers
  - **Precision**: smallest possible difference between any two numbers

The binary point is between bits 2 and 3



- Range: -15.875 to +15.875, Precision: 0.125

Fixed Point Advantages and Disadvantages

- Disadvantage
  - In general, no good way to pick where the fixed binary point should be
    - Sometimes you need range, sometimes you need precision
    - The more of one you have, the less of the other you have
  - Consider a physicist
    - Mass of the sun
      - 1989000000000000000000000000000000 grams - Range (14 bytes)
    - Mass of an electron
      - 0.000000000000000000000000000910956 grams - Precision (12 bytes)
- Advantage
  - Easy to make fast in hardware
    - The same hardware works for both integer and fixed-point arithmetic
    - C does not provide a fixed-point data type, but libraries are available
  - Used extensively in high-performance computing
    - Commercial, scientific, financial applications running on specialized hardware, like FPGAs, DSPs, and ASIPs

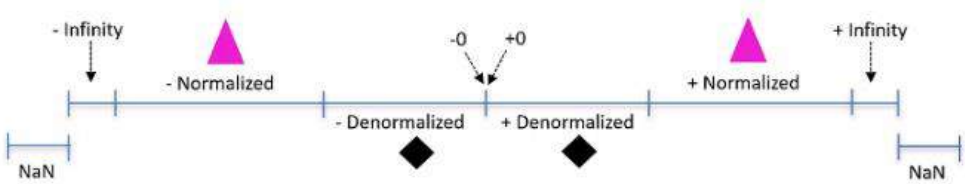
IEEE Floating Point

- Analogous to scientific notation

NOT	BUT	C-Code
1234000000 →	1.234×10^{9}	1.234e9
0.000001234	1.234×10^{-6}	1.234e-6

- IEEE Standard 754
  - Established in 1985 as a standard for floating point arithmetic
    - Before that, many idiosyncratic formats
    - Today, supported by all commodity processors
- Driven by numerical concerns
  - Includes standards for handling overflow, underflow, rounding, etc
  - Hard to make fast in hardware, requires lots of hardware resources

Big Picture: IEEE FP Number Line



Distribution of Values

Floating-Point Representation

- Numerical form
  - Sign bit S determines whether the number is negative or positive
  - Mantissa M is a fractional binary number in the range 1.0 and 2.0 - ε or between 0 and 1.0 - ε
  - The exponent E weights the value by a (possibly negative) value of 2

$$V = (-1)^s \times M \times 2^E$$

- Bit representation
  - MSB is a sign bit S
  - Exponent field encodes E (but is not equal to E)
  - Fraction field encodes M (but is not equal to M)

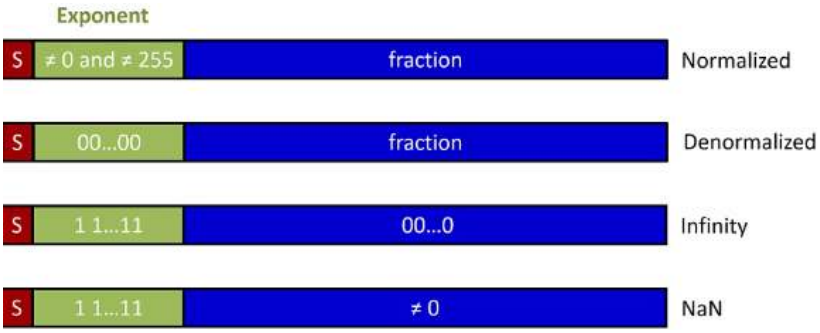
Precisions

- Single precision (32 bits)

- Double Precision (64 bits)

Categories of Floating-Point Values

- The value encoded by a given representation can be divided into three different cases depending on the value of the exponent field



Normalized FP numbers: The Mantissa

- Normalized: means that the mantissa M has the form 1.xxx...x

$$0.011 \times 2^5 = 1100_2$$
$$1.1 \times 2^3 = 1100_2$$

makes better use of available bits

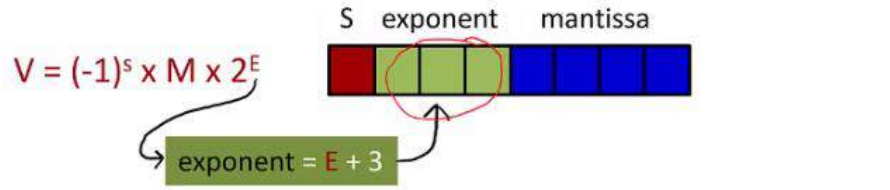
Optimization: since we know the mantissa starts with a 1, there is no need to store it

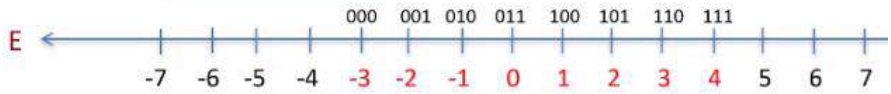
- C code
  - float f = -2345.125;
  - $-2345.125_{10} = -100100101001.001_2$ (equivalent binary number)
  - $= -1.00100101001001 \times 2^{11}$ (normalized form)

Normalized FP numbers: The Exponent

- Normalized: means that the exponents are coded as an unsigned integer in biased form
  - Exponent = E + bias
- Why?
  - So that if we interpret the bit patterns for floating-point numbers as unsigned integers, the floating-point numbers will appear in ascending order for positive numbers and descending order for negative numbers
    - Same (efficient) integer hardware can be used for both signed and unsigned comparisons
    - Floating-point numbers can be sorted using an integer routine

Aside: Consider an 8-bit FP Number



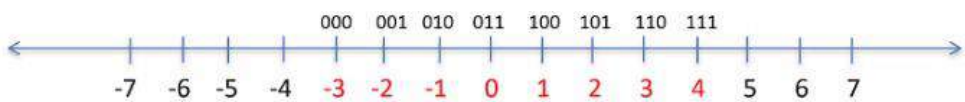


Binary	Unsigned	Signed in biased form	Signed in 2's Complement
000	0	-3	+0
001	1	-2	+1
010	2	-1	+2
011	3	0	+3
100	4	+1	-4
101	5	+2	-3
110	6	+3	-2
111	7	+4	-1

ascending order

of in ascending order

Aside: Comparing Two 8-bit FP Numbers



Binary	Unsigned	Signed in biased form
000	0	-3
001	1	-2
010	2	-1
011	3	0
100	4	+1
101	5	+2
110	6	+3
111	7	+4

- Comparing A and B (=,<,>)

Biased Exponent

- We create a biased exponent by adding a constant to the true exponent
  - $\text{Exponent}_{\text{biased}} = E_{\text{true}} + \text{bias}_{\text{constant}}$
- Where the weight of the constant is $2^{k-1} - 1$ (where k = # bits in exponent)

bias = $2^{K-1} - 1$			
Precision	**bias**	**exponent**	**E = exponent - bias**
Single Precision (k=8)	127	1 to 254	-126 to 127
Double Precision (k=11)	1023	1 to 2046	-1022 to 1023

Summary: Decimal to Floating-Point Conversion

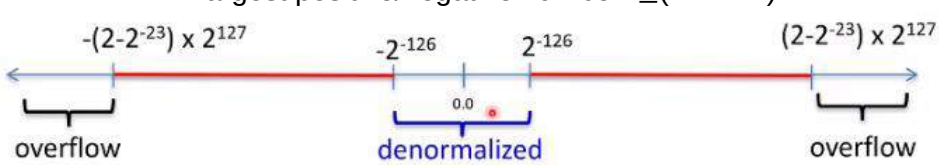
- Number: 347.625_{10}
- Binary 101011011.101×2^0
- Normalize: 1.1011011101×2^8
- Add bias to exponent: 8 + 127 = 135
- Biased exponent to binary: 1000 0111
- Sign bit: 0 if positive, 1 if negative

Floating-Point to Decimal Conversion

- Extract each field and convert to decimal
  - Sign bit (s) is 1_{10}
  - Exponent = E + bias = $01111100_2 = 124_{10}$
  - Mantissa = $1.1100...0_2 = 1.75_{10}$
- Substitute values into $(-1)^s \times M \times 2^{E = \text{exponent - bias}}$
  - $(-1)^1 \times 1.75 \times 2^{124-127} = -1.75 \times 2^{-3} = -0.21875$

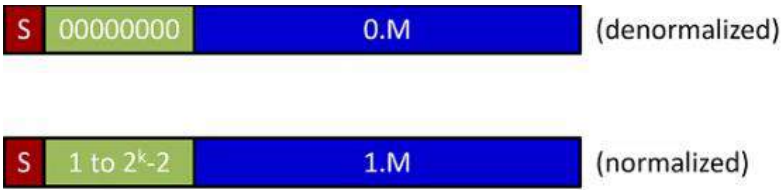
Smallest and Largest Normalized Values

- Single precision
  - Smallest positive/negative number: $\pm(1.00...0) \times 2^{-126} = 2 \times 10^{-38}$
  - Largest positive/negative number: $\pm(1.11...1) \times 2^{127} = 2 \times 10^{38}$
- Double precision
  - Smallest positive/negative number: $\pm(1.00...0) \times 2^{-1022} = 2 \times 10^{-308}$
  - Largest positive/negative number: $\pm(1.11...1) \times 2^{1023} = 2 \times 10^{308}$

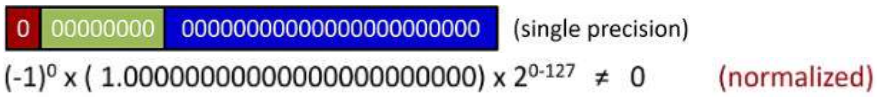


Denormalized Numbers

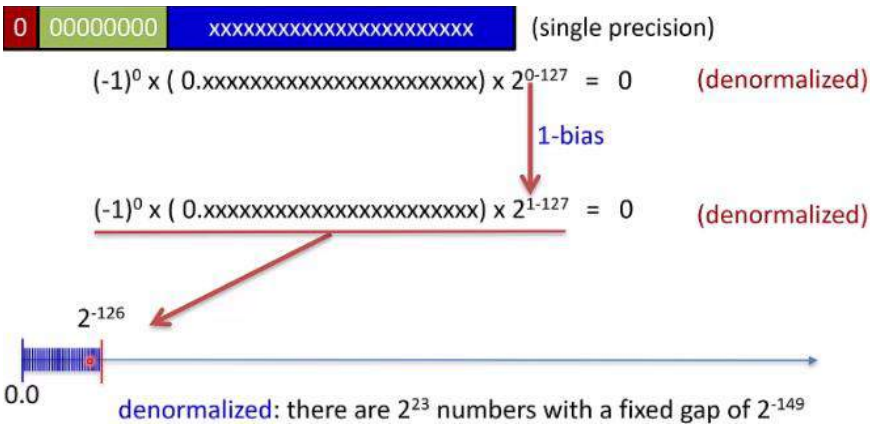
- Denormalized numbers serve two purposes:
  - A way to represent 0
  - A way to represent numbers very close to 0.0 (also known as gradual underflow)
- Assumes a 0 prior to binary point; treat 0 exponent as if it were 1



Representing 0

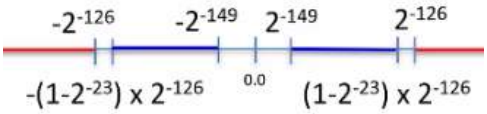


Numbers Close to 0



Smallest and Largest Denormalized Values

- Denormalized (single precision)
  - Smallest positive/negative number: $\pm(2^{-23}) \times 2^{-126} = 2^{-149}$
  - Largest positive/negative number: $\pm(1 - 2^{-23}) \times 2^{126}$
- Between smallest numbers is underflow



Integer vs Floating-Point Numbers

- Why can we seemingly represent many more FP numbers than integers? We cannot!
- With 32-bits, we can represent 2^{32} binary patterns (~4 billion)
  - We can represent 4 billion integers or 4 billion real numbers
  - There are an infinite number of real numbers; the IEEE 754 format can only represent some of them

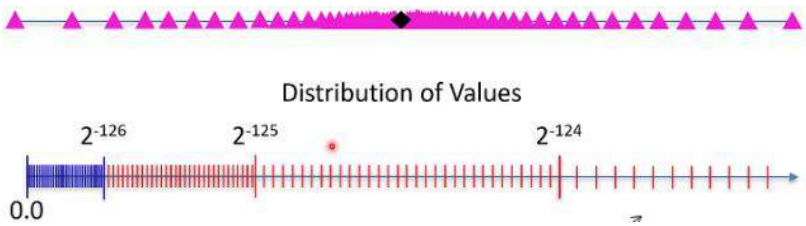
```
int a = 123456789;
float b = 123456789;

printf("%d\n",a);
printf("%f\n",b);
```

→

```
123456789
123456792.00000
```

Gaps Between Floating-Point Numbers

- Unlike the case for integers, the spacing between floating-point numbers is not uniform



Special Cases: Infinity

- Used to represent the value infinity
  - Both positive and negative
  - Captures result of operation that overflows
  - Examples: 1.0 / 0.0 = +inf

Special Cases: NaN

- Used to represent Not-a-Number
  - When no numeric value can be determined
  - Examples: sqrt(-3), inf x 0, inf + inf

Floating-Point Operations

- Step 1: compute the exact result
- Step 2: round the result to make it fit into the desired precision
  - Possibly overflow if exponent too large
  - Possibly drop least-significant bits of significand to fit into mantissa

Rounding

- Different rounding choices

Rounding Choice	$1.35	$1.65	$1.50	$2.50	-$1.50
Toward zero	$1	$1	$1	$2	-$1
Down (-∞)	$1	$1	$1	$2	-$2
Up (+∞)	$2	$2	$2	$3	-$1
To nearest	$1	$2	?	?	?
To even	$2	$2	$2	$2	-$2

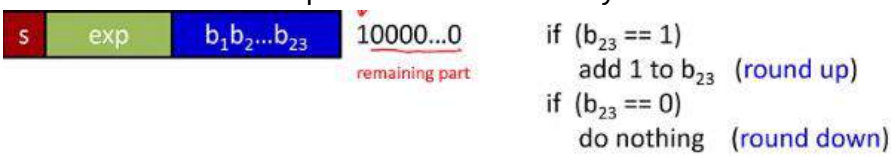
- Problem
  - Rounding in the same direction introduces a statistical bias into our values
  - Round-to-even avoids this bias, and is the default rounding mode for IEEE floating-point

Round-to-Even

- Case 1: Remaining part of binary number is less than ½, where b_{24} is 0
  - Round down by truncating

- Case 2: Remaining part of binary number is greater than ½, where b_{24} & b_{2x} is 1
  - Round up by adding 1

- Case 3: Remaining part of number is equal to ½, where b_{24} is 1 & b_{2x}… is 0
  - Round up or down as necessary



Rounding Binary Numbers to Two Decimal Places

Value	Binary	Rounded	Action	Rounded Value
$2+\frac{3}{32}$	10.00011	10.00	Case 1 ↓	2
$2+\frac{3}{16}$	10.00110	10.01	Case 2 ↑	$2\frac{1}{4}$
$2+\frac{7}{8}$	10.11100	11.00	case 3 ↑	3
$2+\frac{5}{8}$	10.10100	10.10	case 3	$2\frac{1}{2}$

Floating Point Addition

- Procedure
  - Make sure both values are normalized
  - Make the exponents the same (normalize to higher exponent)
  - Add the mantissas
  - If necessary, normalize the sum

X = 0 11010111 11100111010000011000011
Y = 0 11010001 00011100101111100011100

Y = 0.00000100011100101111100 x 2^{215}
X = 1.11100111010000011000011 x 2^{215}

1.11101011101101000111111 x 2^{215}

Exponents

$E_x = 11010111_2 = 215_{10}$
$E_y = 11010001_2 = 209_{10}$

$|E_x - E_y| = 6$

Floating-Point Multiplication

- Procedure
  - Add the (biased) exponents and subtract the bias once
  - Multiply the mantissas
  - Determine sign of product
  - If necessary, normalize the product

X = 1 01111101 00110011001100110011010
Y = 0 10000111 11110100010000000000000

M_X = 1.00110011001100110011010
M_Y = 1.11110100010000000000000

1.00101100001001100110011~~1001011101~~ x 2^{134}

Exponents

$E_x = 01111101_2 = 125_{10}$
$E_y = 10000111_2 = 135_{10}$

$E_x + E_y - 127 = 133$

1	10000110	00101100001001100110100

Single-Precision FP

Distributive, Associative, and Other Properties

- Due to rounding, FP operations are not always associative or distributive

(1.23 + 1e15) – 1e15 ≠ 1.23 + (1e15 – 1e15)
1e25 x (1e25 – 1e25) ≠ (1e25 x 1e25) – (1e25 x 1e25)

1.23 = 0 01111111 00111010111000010100100
1e15 = 0 10110000 11000110101111110101001 Error = -13008896
1e25 = 0 11010010 00001000101100101010001

- Overflow of the exponent
  - Results will be ±inf
- Floating point numbers with values ±inf and NaN can be used in operations

- Usually, results will be still ±inf or NaN

C Programming Language

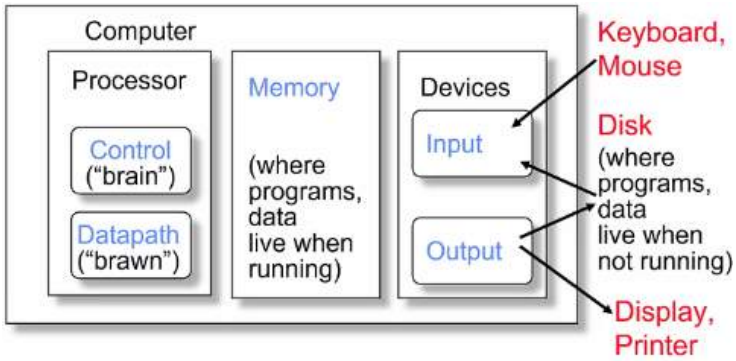
- Guarantees
  - Float (single-precision FP)
  - Double (double-precision FP)
  - Round-to-even
- Type conversion
  - Casting between float, double, and int changes bit representation

Cast Operation	Result
int → float	value may be rounded
int, float → double	exact value preserved
double → float	value can overflow to ± infinity; value may be rounded
float, double → int	truncates fraction; rounds to zero; not defined when overflow occurs or NaN – normally sets to Tmin [1000…0]

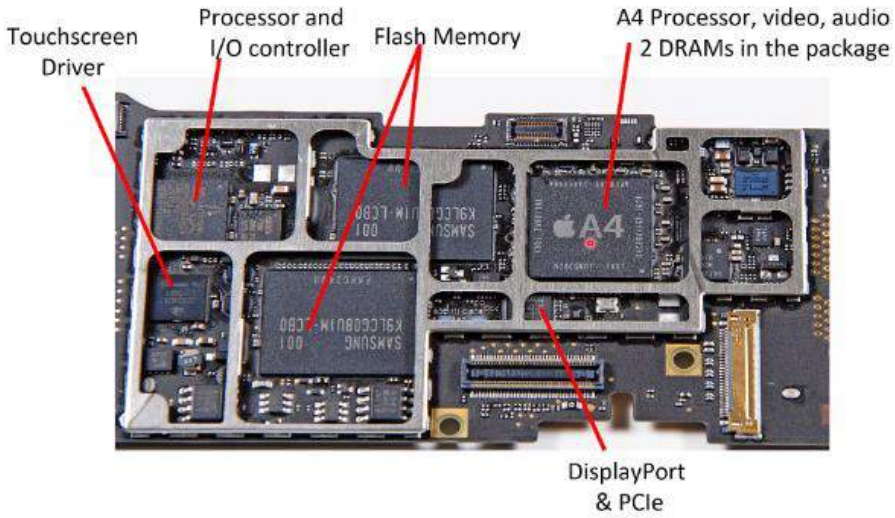
CIS2030 Microcomputers Week 2

Five Components of a Computer

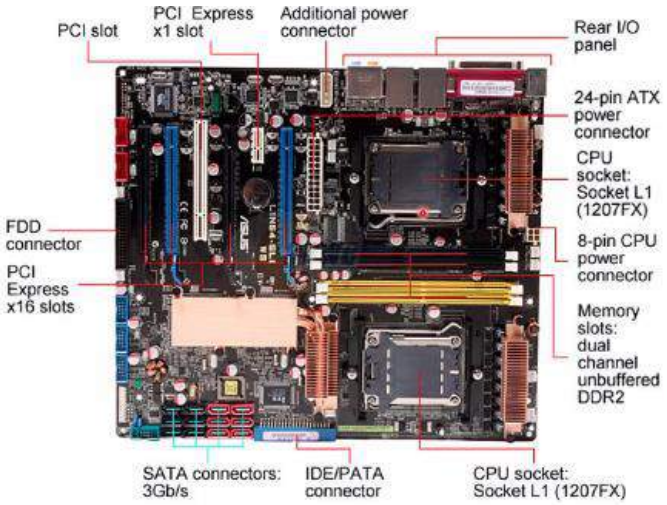
- Any computer, no matter how primitive or advanced, can be divided into five parts:



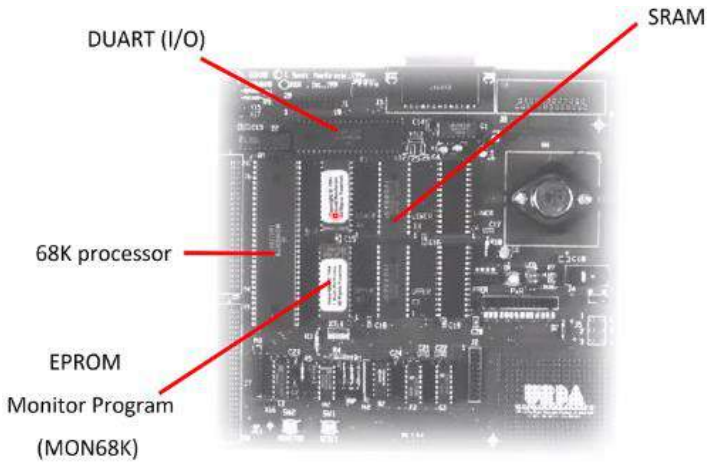
First Example: iPad



Second Example: x86 Motherboard



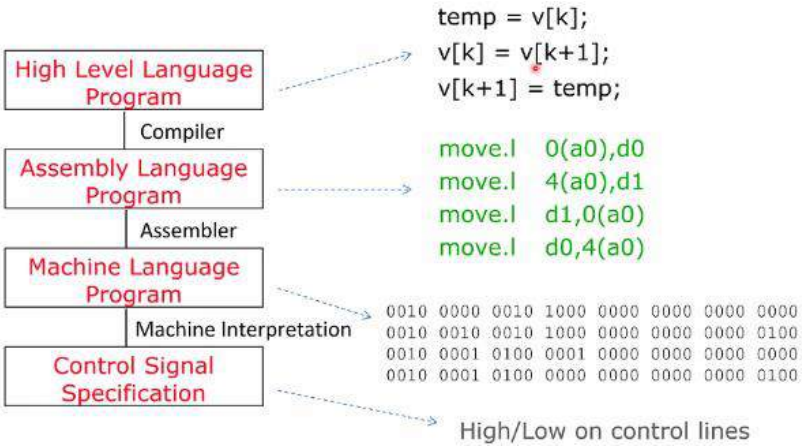
Third Example: 68000 Mini-board (68KMB)



Computers are Pretty Simple

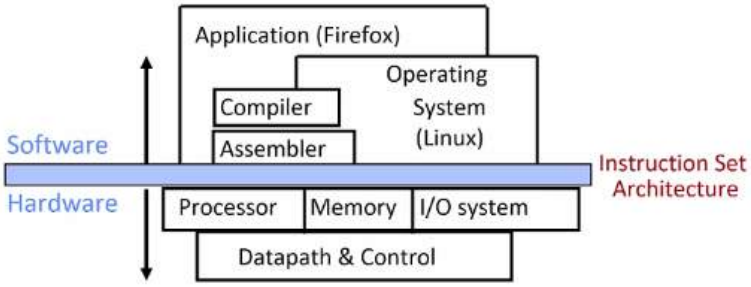
- Computers only work with binary signals
  - Signal on a wire is either 0 or 1
  - More complex stuff (numbers, characters, strings, pictures) must be built from multiple bits
- Built out of simple logic gates that perform Boolean logic
  - AND, OR, NOT, Adders, Comparators, etc
- And memory cells that preserve bits over time
  - Flip-flops, registers, SRAM cells, DRAM cells, Flash, etc
- To get hardware to do anything, need to break it down to bits
  - Strings of bits that tell the hardware what to do are called instructions
  - A sequence of instructions is called machine code or machine-language program

Running an Application



Instruction-Set Architecture (ISA)

- Instruction-Set Architecture
  - Important abstraction
  - Interface between hardware and lowest-level software
  - Portion of computer visible to programmer
    - Assembly language, programmer's model, instructions/format



A First Look at Memory

- Memory holds both **instructions** and **data**
- Logically organized as an array of **locations**, each with an **address**
- Process
  - Running program + State
  - Each process "sees" full address space due to an abstraction, called virtual memory
- Compiler + run-time system
  - Control memory allocation

Word Size of Processor

- Nominal size of integer-valued data
  - Includes addresses used to index memory
- Until recently, most machines used 32-bit (4-byte) words
  - Limits addresses to 4 Gigabytes
  - Become too small for memory-intensive applications
- Most current machines use 64-bit (8-byte) words
  - Potential address space: $2^{64} \approx 1.8 \times 10^{19}$ bytes (16 exabytes)
- For backward-compatibility many processors support different word sizes
  - Always a power-of-2 in the number of bytes: 1, 2, 3, 4, 5, 6, …

C Data Types

- Typical size of C objects in bytes

Data Object	32-bit Machine	64-bit Machine
char	1 byte	1 byte
int	4 bytes	4 bytes
short int	2 bytes	2 bytes
long int	4 bytes	8 bytes
long long	8 bytes	8 bytes
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	8 bytes	16 bytes
pointer	4 bytes	8 bytes

Word-Oriented View of Memory

- Addresses specify locations of bytes in memory
    - Word address is address of first byte in word
    - Address of successive words differ by 4 (32-bit) or 8 (64-bit)

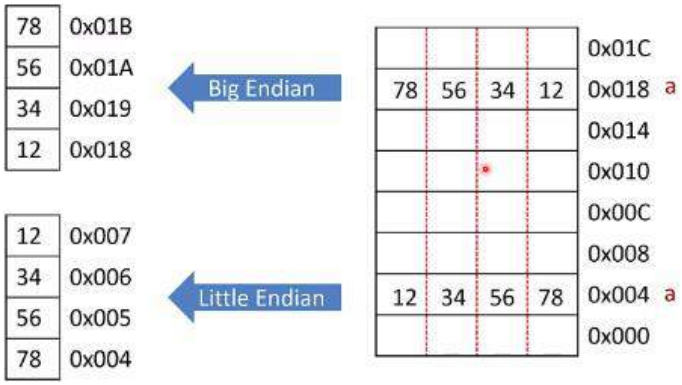
Variable Declarations and Memory

- Variable declarations
    - int a,b;
    - Find two locations in memory in which to store two integers (1 word each)
- Memory layout
    - Assume word size of 32-bits
- Assignment (LHS = RHS)
    - int a = 0, b = -1;
    - LHS must evaluate to a memory location (a variable)
    - RHS must evaluate to a value (possibly an address)
- Pointer declaration
    - int *a;
    - Declares a variable a that is a pointer to an integer data item
- Pointer assignment
    - a = &b;
    - Assigns a the address where b is stored

- Pointer dereferencing
  - *a;
  - The value at the memory address given by the value of *a
- Pointer arithmetic
  - a = a + 1;
  - The value of the pointer is incremented to point at the next data object (integer) in memory

Byte Ordering

- How do we order the bytes in a word?
  - int a = 0x12345678;

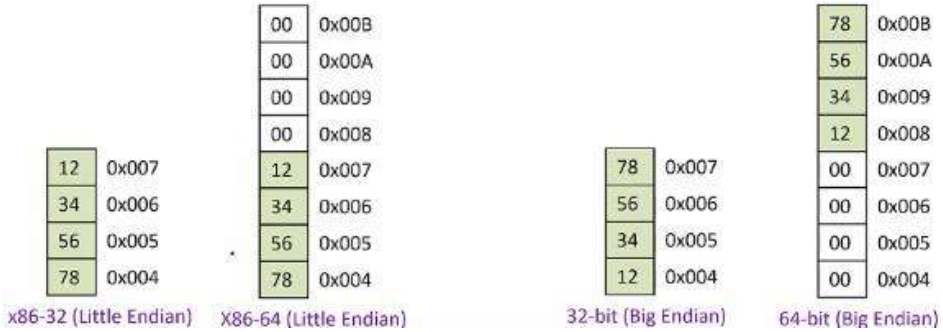


Endianness: Big or Little?

- Big Endian
  - Address of most-significant byte = (lowest) address of word
  - IBM 360, M68000, MIPS, SPARC
- Little Endian
  - Address of least-significant byte = (lowest) address of word
  - Intel x86, ARM, DEC Vax & Alpha

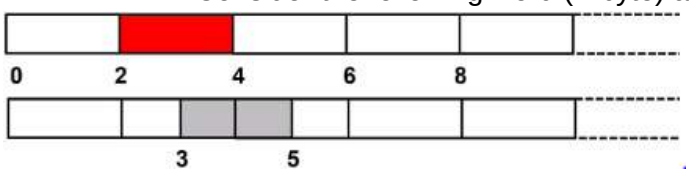
When Endianness Matters

- When you store words then load bytes
- When you share data between machines with different endianness
- When moving between 32-bit and 64-bit word sizes



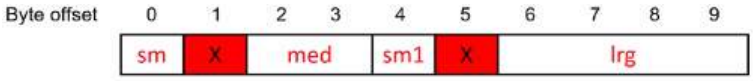
Alignment Issues

- Some ISAs require words to be at an even address
  - Consider the following word (2-byte) access



- C example

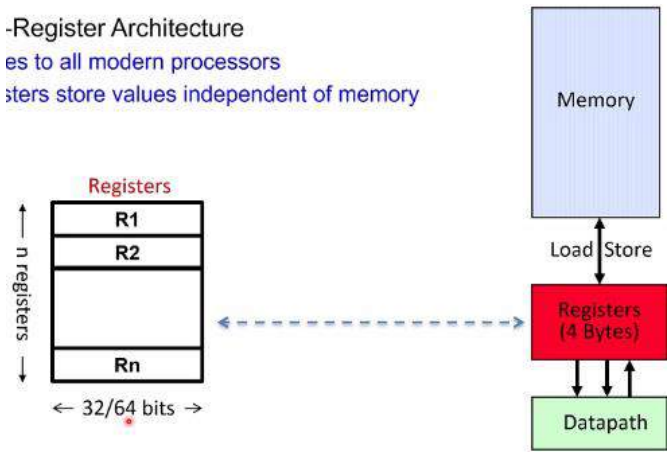
```
struct foo {
    char   sm;     /* 1 byte  */
    short  med;    /* 2 bytes */
    char   sm1;    /* 1 byte  */
    int    lrg;    /* 4 bytes */
}
```



- Historically
  - Early machines (IBM 360, circa 1964) required alignment
  - Removed in the 1970s to reduce impact on programmers (IBM 370, x86)
  - Reintroduced by RISC to improve performance
  - Removed by some RISCs to simplify software

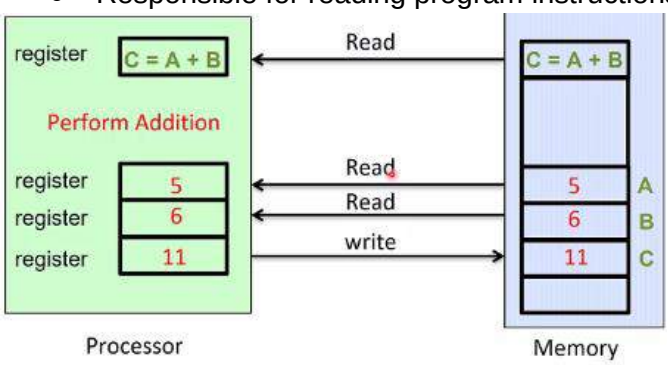
A First Look at Registers

- General-Register Architecture
  - Applies to all modern processors
  - Registers store values independent of memory

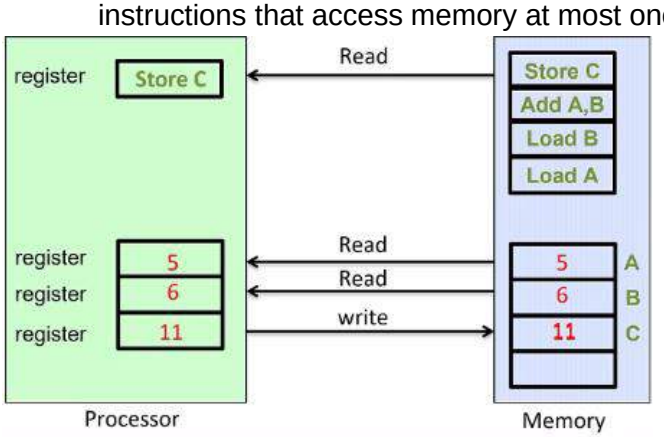


A First Look at the Processor

- Responsible for reading program instructions from memory and executing them

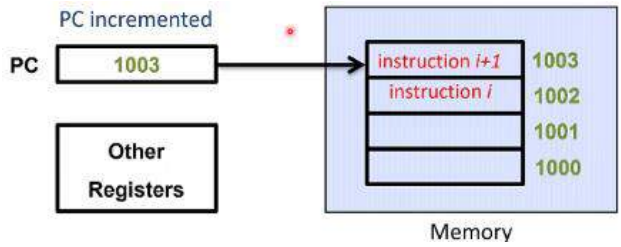


- A statement such as C = A + B is implemented as a sequence of primitive machine instructions that access memory at most once



Program Counter

- PC is a **register** inside the processor
- PC holds the address of the next instruction to be fetched and executed



Instruction Cycle

- Set PC = address of first instruction in program
- Repeat
    - **Fetch** the instruction
        - Get an instruction from memory location contained in PC
        - Increment PC by size of instruction
    - **Decode** the instruction
        - Identify operation defined by instruction
        - If instruction requires data from memory, fetch data from memory
    - **Execute** the instruction
        - Perform operation defined by instruction
        - If instruction requires data to be stored, store data in memory
- Until (last instruction encountered)

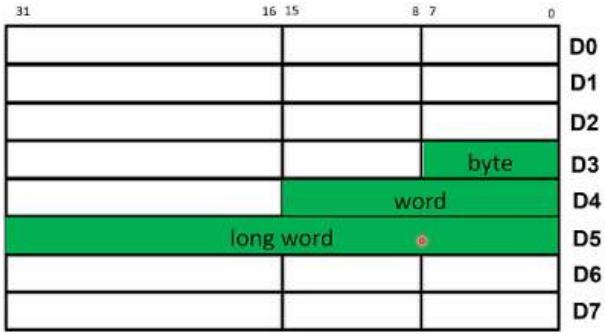
C vs M68000 Programmer's Interface

	C	M68000 ISA
Registers		8, 32b integer, D0-D7 7, 32b address, A0-A6 2, 32b stack pointers, USP & SSP SR, 16b PC, 24b
Memory	local variables global variables	2^{24} linear array of bytes
Data types	int, short, char, unsigned, float, double, aggregate data types, pointers	byte(8b), word (16b) and long-word (32b)
Arithmetic operators	+, -, *, %,++, <, etc.	add, sub, mul, div, etc.
Memory access	a, *a, a[i], a[i][j], a.x	13 address modes
Control	If-else, while, do-while, for, switch, function call, return	bcc, jmp, rts, link, unlink

Data Registers

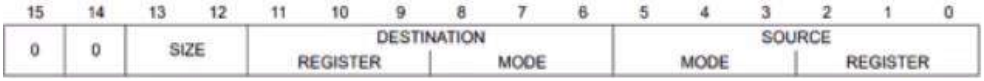
- Data registers are used to hold data independent of memory
    - 8, 32-bit homogeneous registers

- Allow byte, word, and long-word operations



MOVE

- Copy data from source to destination
- Operation: source → destination
- Syntax: MOVE <ea>,<ea>
  - EA means effective address
- Sample syntax:
  - MOVE (A5),-(A2)
  - MOVE -(A5),(A2)+
  - MOVE Temp1,Temp2
- Attributes: Size = byte, word, longword, . is used as size indicator
  - MOVE.B D0, D1 (1st byte)
  - MOVE.W D0, D1 (word, 1st & 2nd bytes)
  - MOVE.L D0, D1 (longword, all 4 by 1)
  - MOVE D0, D1 (default is .W)
- Description: Move the contents of the source to the destination location. The data is examined as it is moved and the condition codes set accordingly. Note that this is actually a copy command because the source is not affected by the move. The move instruction has the widest range of addressing modes of all the 68000's instructions
- Condition Codes:
  - X - not affected (-)
  - N - set if the result is negative; cleared otherwise (*)
  - Z - set if the result is zero; cleared otherwise (*)
  - V - always cleared (0)
  - C - always cleared (0)
- Instruction Format:



- Instruction Fields:
  - Size field: specifies the size of the operand to be moved
    - 01 - byte operation
    - 11 - word operation
    - 10 - long operation

- Destination Effective Address field: specifies the destination location. Only data alterable addressing modes can be used as listed in the following table:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—

- Source Effective Address field: specifies the source operand. All addressing modes can be used as listed in the following table:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An*	001	reg. number:An	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	111	100
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011

Copying Data Between Data Registers

- Copy the byte from D0 to D1
  - move.b d0,d1 → D0 12 34 56 78 / D1 00 00 00 78
- Copy the word from D0 to D1
  - move.w d0,d1 → D0 12 34 56 78 / D1 00 00 56 78
- Copy the long-word from D0 to D1
  - move.l d0,d1 → D0 12 34 56 78 / D1 12 34 56 78

ADD

- Add binary
- Operation: [destination] ← [source] + [destination]
- Syntax:
  - ADD <ea>,Dn
  - ADD Dn,<ea>
- Attributes: Size = byte, word, longword
- Description: Add the source operand to the destination operand and store the result in the destination location

- Condition Codes:
    - X - set the same as the carry bit (*)
    - N - set if the result is negative; cleared otherwise (*)
    - Z - set if the result is zero; cleared otherwise (*)
    - V - set if an overflow is generated; cleared otherwise (*)
    - C - set if a carry is generated; cleared otherwise (*)

C Example Involving Addition

- Consider the C operation for addition
    - a = a + b;
- Assume that the variables are bytes (i.e., chars) and a is contained in D1 and b is contained in D0
- To perform the addition use the **ADD** instruction
    - add.b d0,d1 ;a = a + b

SUB

- Subtract
- Operation: destination - source → destination
- Syntax:
    - SUB <ea>,Dn
    - SUB Dn,<ea>
- Attributes: Size = byte, word, longword
- Description: Subtracts the source operand from the destination operand and stores the result in the destination. The size of the operation is specified as byte, word, or long. The mode of the instruction indicates which operand is the source, which is the destination, and which is the operand size
- Condition Codes:
    - X - set to the value of the carry bit (*)
    - N - set if the result is negative; cleared otherwise (*)
    - Z - set if the result is zero; cleared otherwise (*)
    - V - set if an overflow is generated; cleared otherwise (*)
    - C - set if a borrow is generated; cleared otherwise (*)

Complex Operations

- What about more complex statements?
    - a = a + b + c - d

- Break into multiple instructions
- Assume a(D0), b(D1), c(D2), d(D3) are bytes
    - add.b d1,d0 ;a = a + b
    - add.b d2,d0 ;a = (a + b) + c
    - add.b d3,d0 ;a = (a + b + c) - d

Constant Values

- Consider the following C code
    - a = 3;
- Often want to be able to specify a constant in an instruction
    - These are called immediate or literal values
- Use the # symbol to specify a constant
    - move.b #3,d0 ;a = 3
- The immediate is an 8-bit, 16-bit or 32-bit value and must not exceed the range of the size indicator
- The immediate value can be specified in hex($), binary(%), or decimal (nothing)

Example the word in D0 with the constant 10

- move.w #10,d0
- move.w #$A,d0
- move.w #$1010,d0

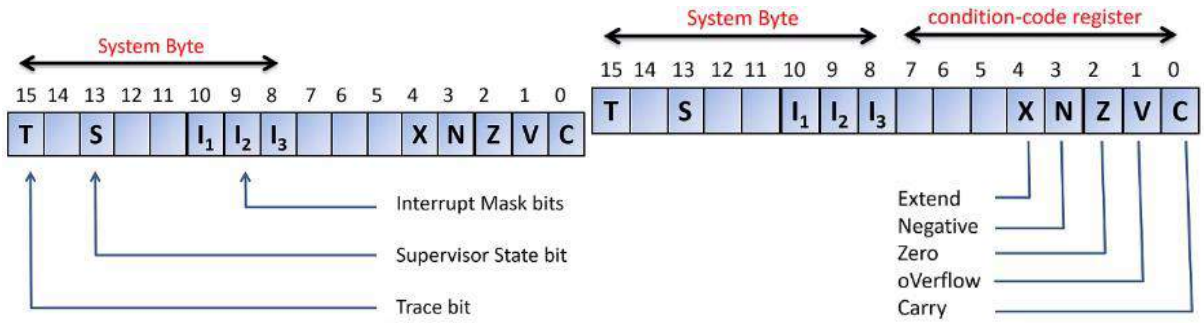
68000 Simple Arithmetic

Instruction	Size	Example	Meaning
Addition	B/W/L	add d0,d1	d1 = d1 + d0
Subtraction	B/W/L	sub d0,d1	d1 = d1 – d0
Multiplication unsigned	W	mulu d0,d1	d1 = d0 x d1; (32-bit result)
Multiplication signed	W	muls d0,d1	d1 = d0 x d1; (32-bit result)
Division unsigned	L	divu d0,d1	d1[15:0] = d1[31:0] / d0[15:0] d1[31:16] = d1[31:0] % d0[15:0]
Division signed	L	divs d0,d1	d1[15:0] = d1[31:0] / d0[15:0] d1[31:16] = d1[31:0] % d0[15:0]
Swap	L	swap d1	d1[15:0] = d1[31:16] d1[31:16] = d1[15:0]
Sign Extension	W/L	ext d1	d1[15:8] = d1[7] d1[31:16] = d1[15]

EXT

- Sign extend
- Operation: destination sign-extended → destination
- Assembler: EXT.W Dn (extend byte to word)
- Syntax: EXT.L Dn (extend word to long word)
- Attributes: size = word, long
- Description: extends a byte in a data register to a word or a long word, or a word in a data register to a long word, by replicating the sign bit to the left. If the operation extends a byte to a word, bit 7 of the designated data register is copied to bits 15 - 8 of that data register. If the operation extends a word to a long word, bit 15 of the designated data register is copied to bits 31 - 16 of the data register

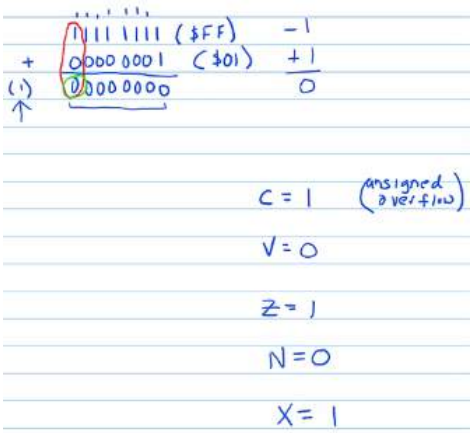
Status Register



- The condition-code register is updated after most instructions execute to reflect the characteristics of the result

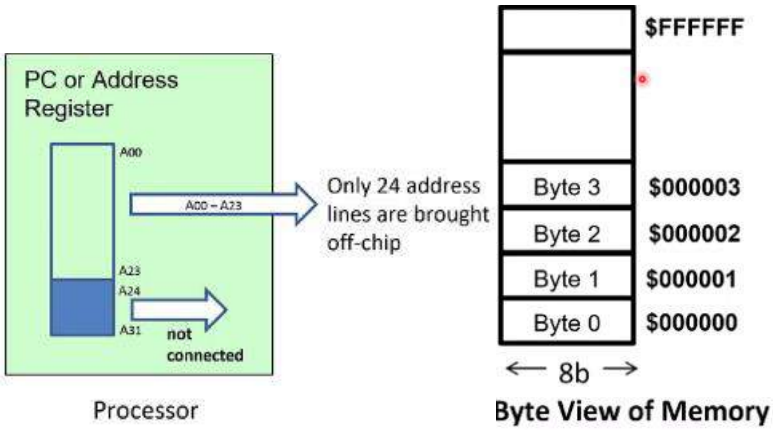
Examples

- What are the contents of the condition-code register after the following program segment executes
    - move.b #$ff,d0
    - add.b #1,d0



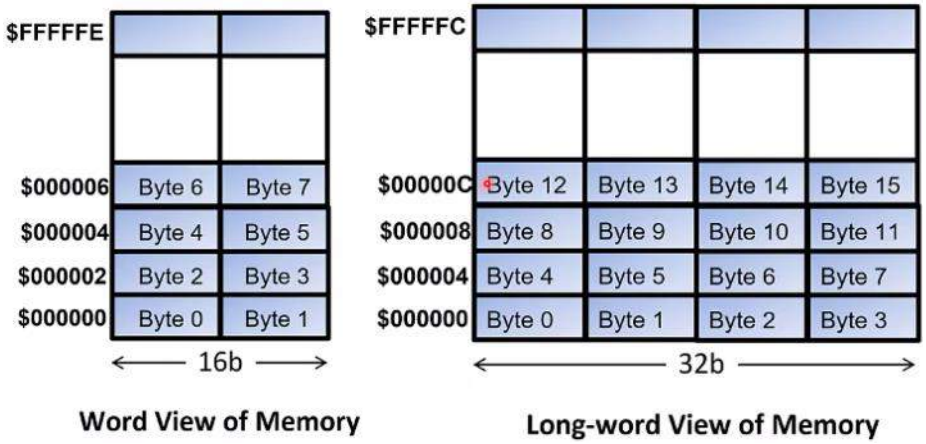
Memory

- The M68000 has a 24-bit address bus which limits the size of the addressable memory to 16 MB



Word and Long-Word Views of Memory

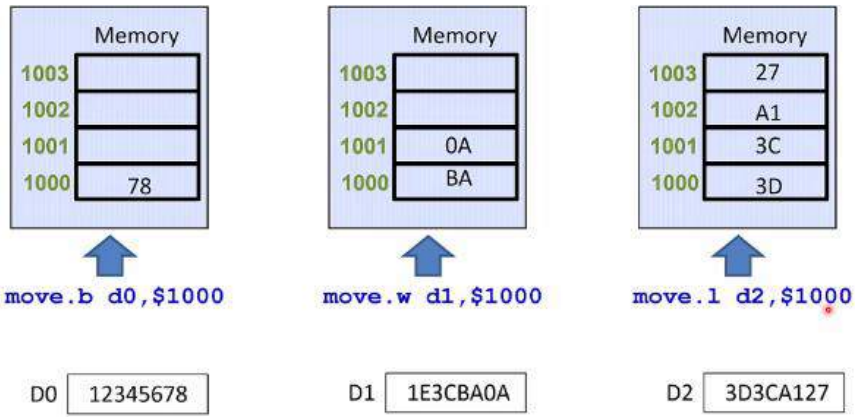
- The M68000 also supports word (2-byte) and long-word (4-byte) operations on memory



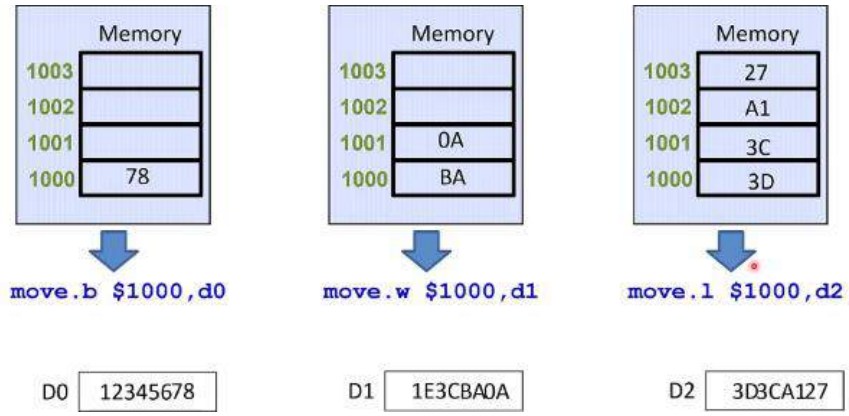
Endianness: Big or Little?

- Big Endian
  - Address of most-significant byte = address of word/long-word

Examples: Storing Data in Memory



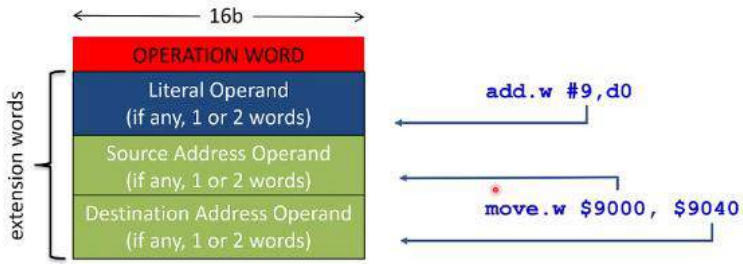
Examples: Loading Data in Memory



Instruction Formats

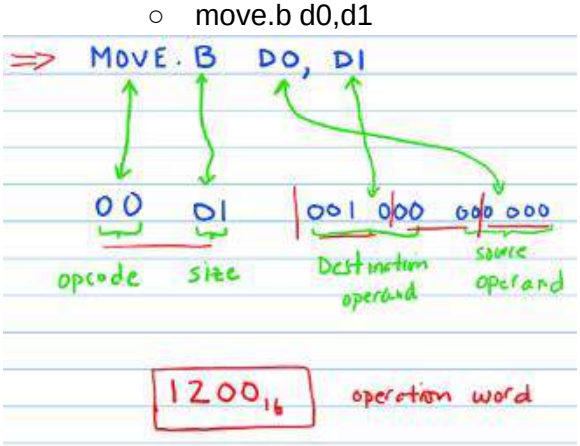
- A M68000 instruction consists of one to five words (16 bits)
  - An instruction may have 0, 1, or 2 operands

- Operands may be bytes, words, or long-words
- An instruction may access memory 0, 1, or 2 times
- Destination operands are changed by the execution of an instruction
- Source operands are not changed by the execution of an instruction



In-Class Examples

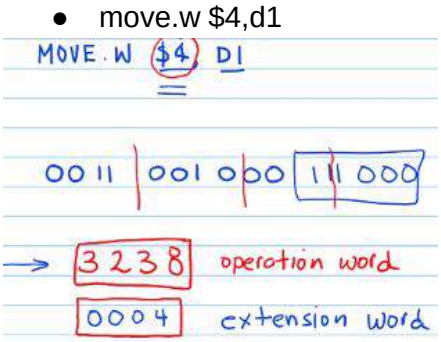
- Hand assemble the following M68000 instructions
  - move.b d0,d1



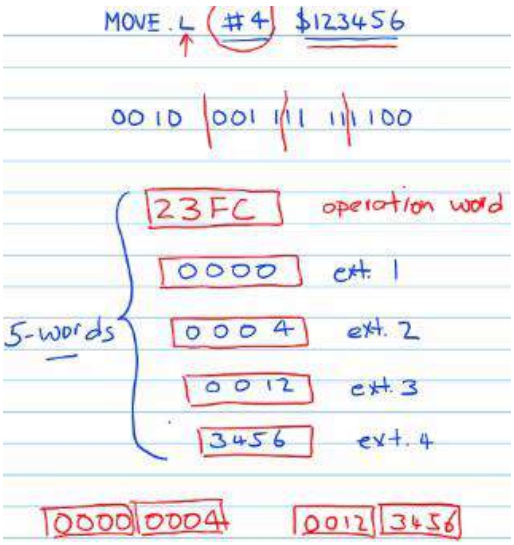
- Assembler does this automatically

```
00001000  1200                    10          MOVE.B  D0,D1
```

- move.w $4,d1



- move.l #4,$123456



Easy68K

- The origin assembler directive (ORG) specifies where to start loading program instructions (or data) into the target processor's memory space
    - ORG $1000 specifies that the program instructions that follow in the source file are to be loaded into memory starting at memory address 0x00001000 (the prefix $ indicates that the address is specified as a hexadecimal number)
- Lines that begin with an * are full-line comments, these must begin at line 1 to avoid syntax errors during the assembly process, while in-line comments are started with a ;
- START is a label, programmers use labels to reference a particular line of code or a particular memory address, in this case, START is referenced by the END START assembler directive, and is used to indicate where (in memory) the program should start execution
- LEA loads the value associated with the label MSG into address register A1
- TRAP instructions provide a way for a program to request services from the operating system, TRAP 15 is used to perform input-output tasks built into the sim
- The assembler directive define constant (DC) is used to load an ASCII string into memory at current location
- SIMHALT is an assembler directive that causes the sim to halt execution

CIS2030 Microcomputers Week 3

Assembly-Language Statements

- Assembly language is made up of two types of statements
    - Executable statements
        - Valid processor instructions

        - Translated into machine language for the assembler
    - Assembler Directives
        - Are not instructions to the processor
        - Cannot be translated into machine language
        - Tell the assembler what it needs to know about the program and its environment

68000 Program Structure

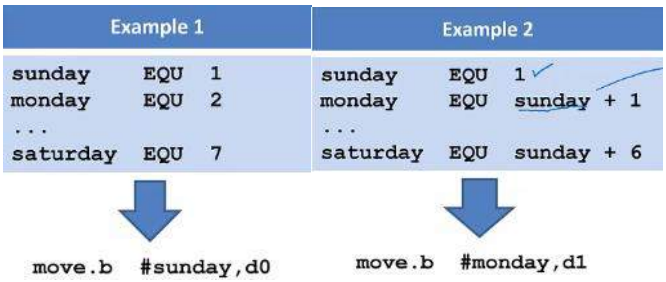
- Assembly language programs are organized according to columns
    - Label Field (column 1)
        - Labels in column 2 cause an error
        - Are optional
    - MNEMONIC
        - Instructions and assembler directives
    - Operands
        - Data required by MNEMONIC column
    - Comment
        - Requires a semicolon ;
        - Assembler ignores these

The Most Important Assembler Directives

Directive	Operation	Syntax
ORG	Set value (address) of location counter	`ORG address`
END	End of source program	`END label`
EQU	Equate value to symbol	`symbol EQU value`
DS	Reserve space in RAM for data generated at runtime	`<label> DS count`
DC	Define data in RAM at the current location	`<label> DC item, <item> ...`

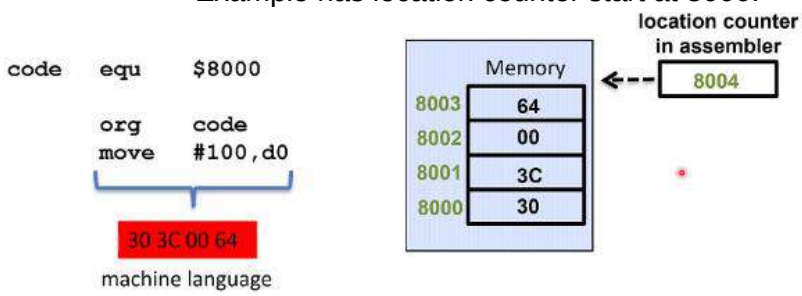
Equate Directive

- Links a name to a value, making programs easier to read
    - Format: <name> EQU <value> (Does not reserve space in RAM)
    - The RHS can contain an assembly-time expression as long as all of the symbols have been pre-defined



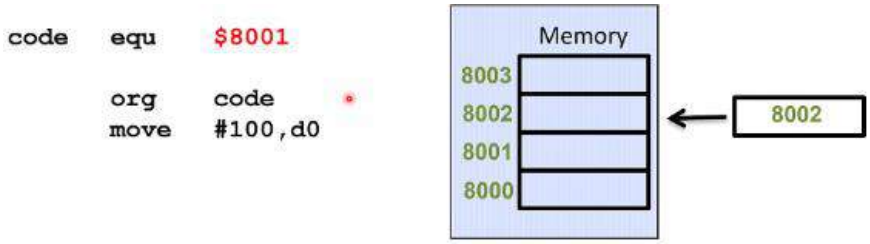
Origin Directive

- Sets up the value of the location counter and keeps track of where the next item is to be located in the processor's memory
    - Format: ORG <address> (location counter = address)
    - Example has location counter start at 8000:



Alignment and the Location Counter

- If an odd number address is specified, the counter bumps up by 1

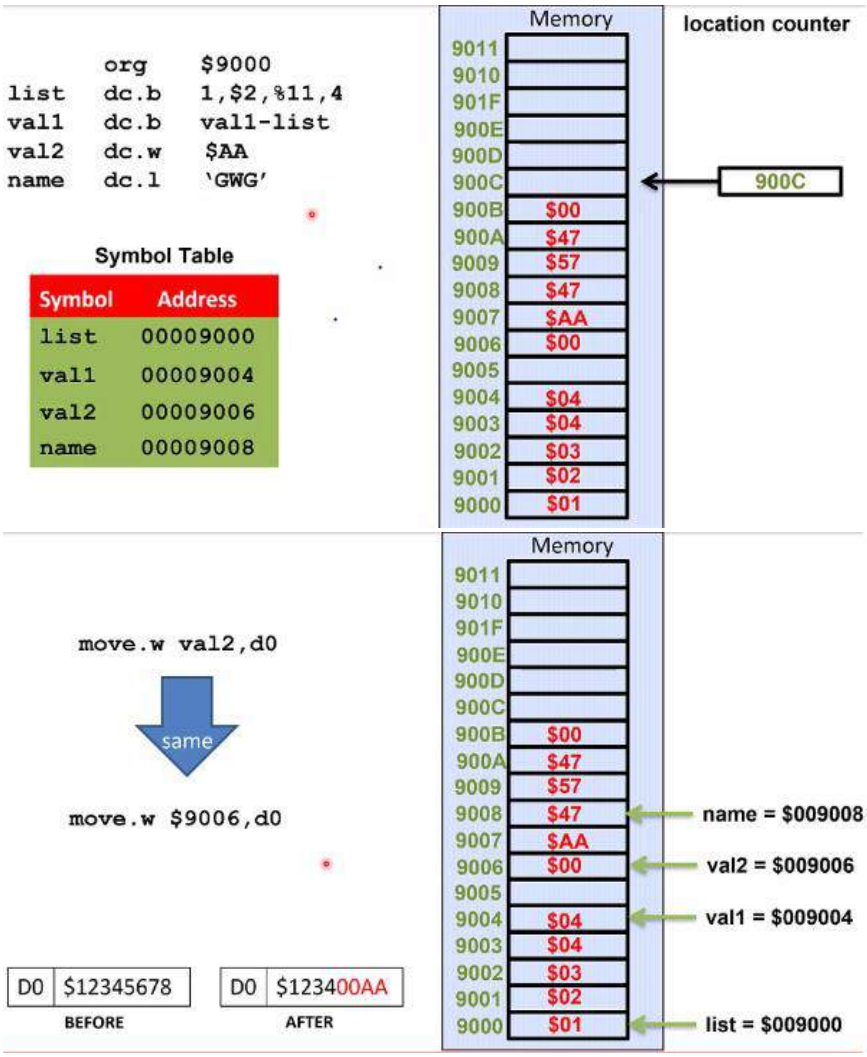


- An ORG directive can be located at any point in the program
- The M68000 ISA reserves the first 1024 bytes of memory
- RAM starts at hexadecimal $8000 on the 68K mini-board

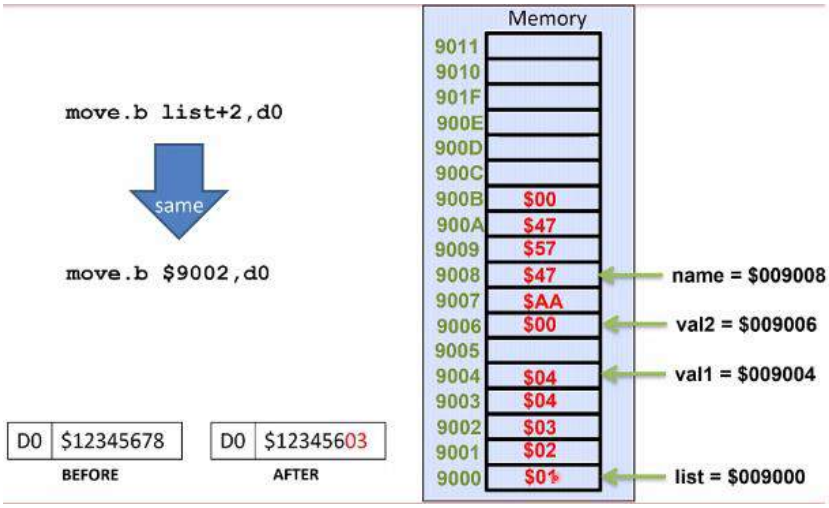
Define Constant Directive

- The define constant directive allows you to place one or more data items in memory when the program is first loaded
    - Format: <label> DC.<size> <item>,<item> ...

- The (optional) label will be defined to equal the address of the start of the list of items
- The size specifies that a list of bytes (.B), words (.W), or long-words (.L) is being defined
- Each item may be a decimal number, hexadecimal number ($), a binary (%) number, an ASCII string enclosed in single quotes, or an assemble-time expression



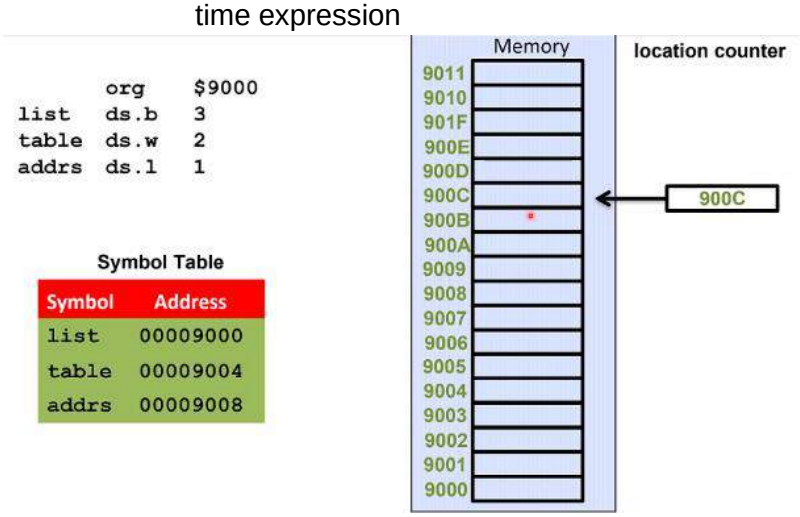
- Common mistake: don't put #val2 as it uses the address of val2 (9006)
- Assemble-time expressions and labels

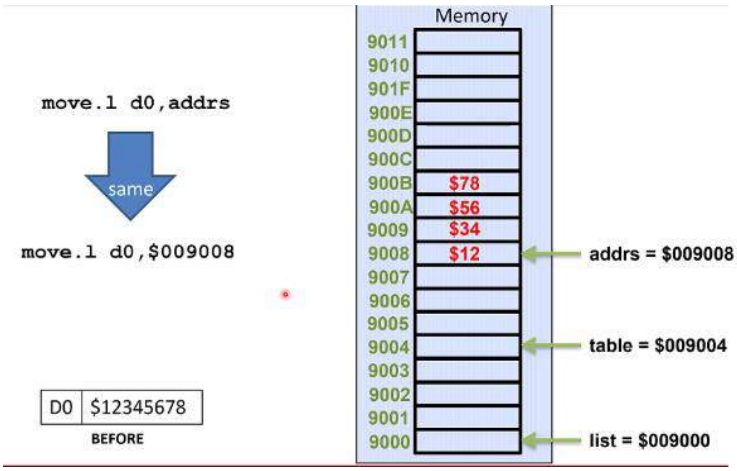


- Refer to location counter directly using *
    - Instead of val1 dc.b val1 - list, do dc.b * - list

Define Storage Directive

- The define storage directive allows you to reserve an uninitialized block of bytes, words, or long-words in memory
    - Format: <label> DS.<size> <count>
    - The label will be defined to equal to the address of the start of the memory block
    - The size specifies that a block of bytes (.b), words (.W), or long-words (.L) is being reserved
    - count is the number of bytes, words, or long-words that will be in the block, and may be specified as a decimal number, hexadecimal, a binary, or an assemble-time expression

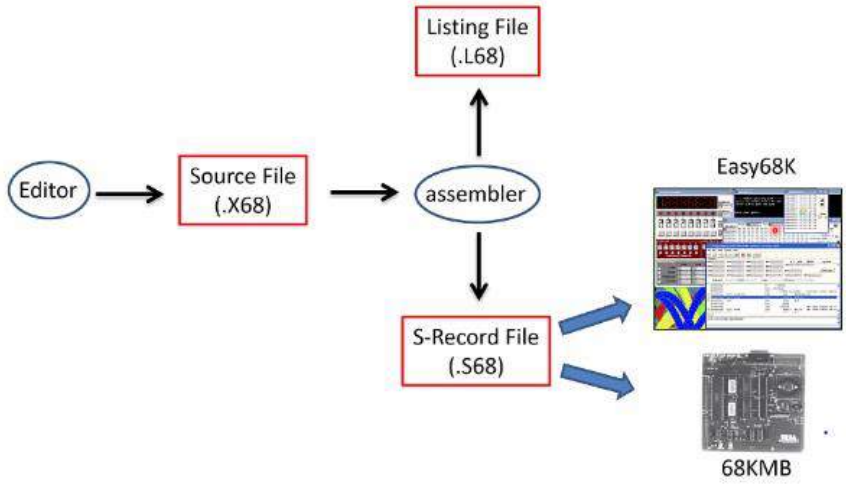




End Directive

- The end directive marks the end of the source file
    - Format: END <start address of program>
    - END should be at the end of code (doesn't have to be though)

Assembler Flow



CIS2030 Microcomputers Week 4

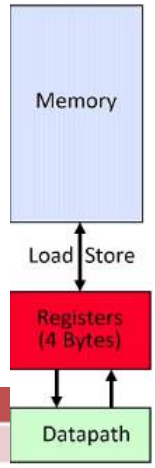
Address Modes

- An address mode refers to the way in which an operand's location is specified
- Some examples that we have already seen include

    - ADD.W D0,D1
    - ADD.W $9000,D1
    - ADD.W #$9000,D1
- The 68000 supports 13 unique address modes
    - Not all address modes can be used with all instructions/operands
- Why support more than one address mode?
    - In theory one address mode suffices, but
    - More powerful address modes increases the efficiency of programs

Data Register Direct

- Operand is contained inside a data register
    - Fastest address mode
    - Shortest instructions
    - Typically used for temporary values and frequently accessed variables



Assembler Instruction	Machine Language
move.b d0,d3	1600

C and Data-Register Direct Addressing

- The C register keyword can be used as a hint to the compiler that the variable should be stored in a register rather than RAM
- Consider the following C code

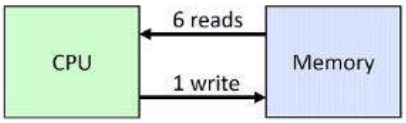
```
for(register int i=0; i < bufsize; i++) {

    // do something
}
```

Absolute Long Addressing

- Operand is in memory and address of operand is contained in two extension words as part of the instruction

- Slowest address mode
- Longest instructions
- Used to access simple global variables

Assembler Instruction	Machine Language
move.b $9000,$9001	13F9 0000 9000 0000 9001

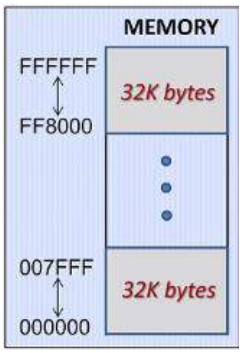


Absolute Short Addressing

- Operand is in memory and address of operand is contained in one extension word as part of the instruction

Assembler Instruction	Machine Language
move.w $ff8000,d0	2038 8000

Assembler Instruction	Machine Language
move.w $007fff,d0	2038 7fff



- To override absolute short addressing add a .L at $7000 for example

Immediate Addressing

- Operand is contained inside the instruction using one or two extension words
  - Specified using # symbol
  - Used for all true constants and to initialize memory and registers
  - Can only be used to specify a source operand

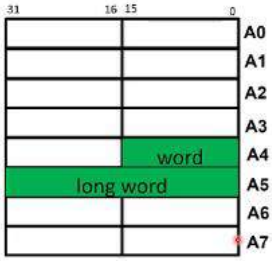
Assembler Instruction	Machine Language	Result
move.l #$ff,d0	203C 0000 00FF	D0 = $000000FF
move.w #$ff,d0	303C 00FF	D0 = $----00FF
move.b #$ff,d0	103C 00FF	D0 = $------FF

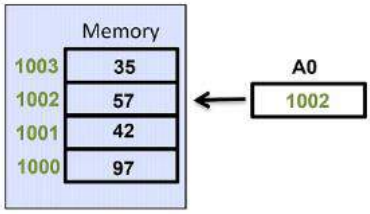
- MOVE.L #$77,D0 uses MOVEQ where it sign extends, and can get away with it, unlike #$88

Address Registers

- Address registers are used to hold the address of a memory location
  - Eight address registers, each 32 bits

- Byte operations not allowed
- Special one: A7 is used a stack pointer





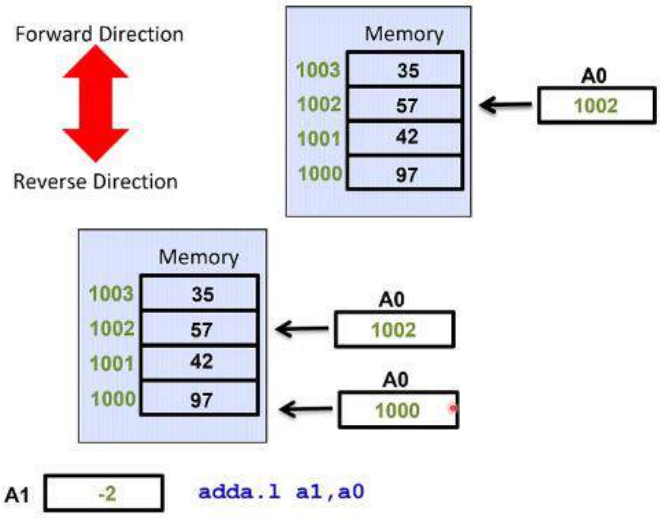
Memory[A0] = Memory[1002] = 57

Address Registers != Data Registers

- If an address register is specified as the destination operand, the following instructions must be used
  - MOVEA, ADDA, SUBA (CCR flags not affected)
- Data registers use the following instructions
  - MOVE, ADD, SUB (CCR flags affected)

Instruction	Correct (Y/N)?
`adda.l d3,a3`	Y
`add.l d3,a3`	N
`add.l a3,d3`	Y

- Address stored in an address register is considered to be a 32-bit signed entity



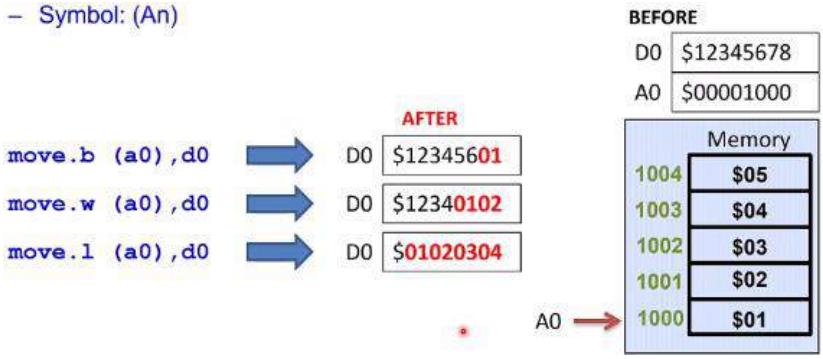
Sign-Extension of Word Addresses

- All addresses are sign-extended to 32-bits for word operations
  - adda.l #$fff4,a0 a0 = a0 + $0000fff4
    - Machine language: D1FC 0000 FFF4
  - adda.w #$fff4,a0 a0 = a0 + $ffffff4

- D0FC FFF4

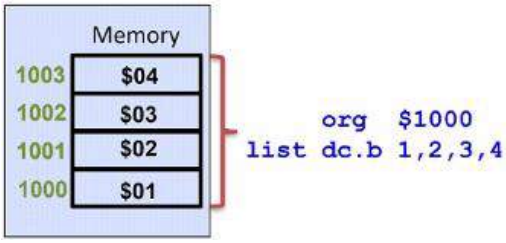
Indirect Addressing

- Operand is in memory and address of operand is contained inside an address register
  - Symbol: (An)



Indirect Addressing and Pointers in C

- Consider the example
  - ptr = &list;
  - first = *ptr;



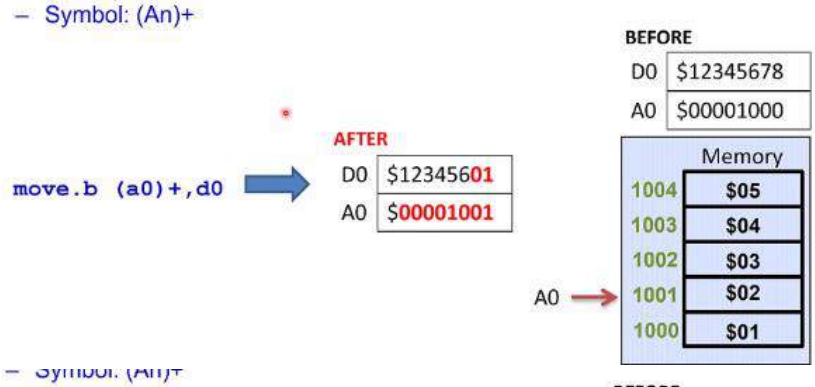
- Assume that the list variables are bytes (i.e., chars) and first is contained in D0 and ptr is to be implemented using A0

```
movea.l #list,a0        ;a0 = 1000
move.b  (a0),d0         ;d0 = memory[1000] = 1
```

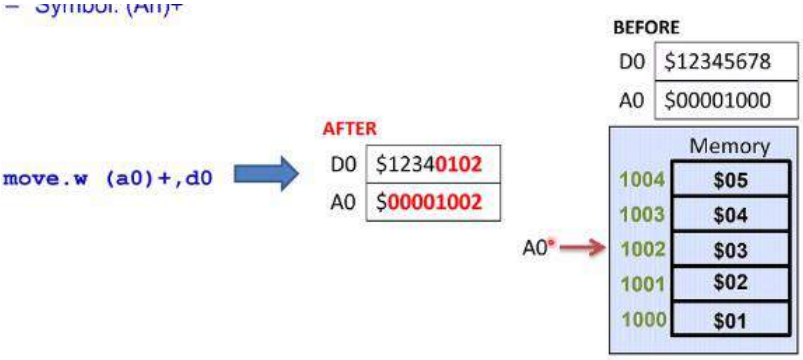
Post Increment Indirect Addressing

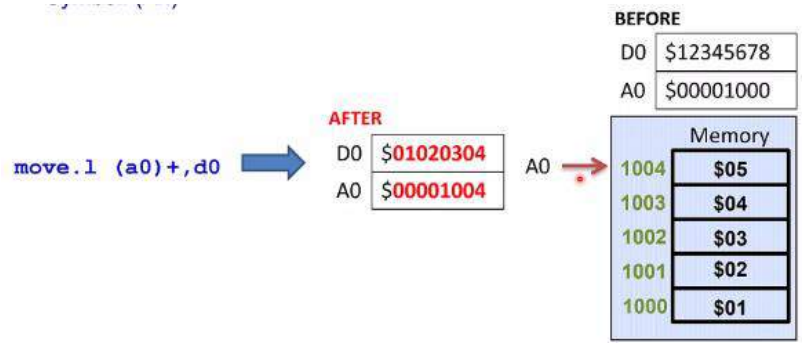
- Same as indirect, but after the operand is used, the address in the register is incremented by the size of the operand (in bytes)

– Symbol: (An)+



– Symbol: (An)+

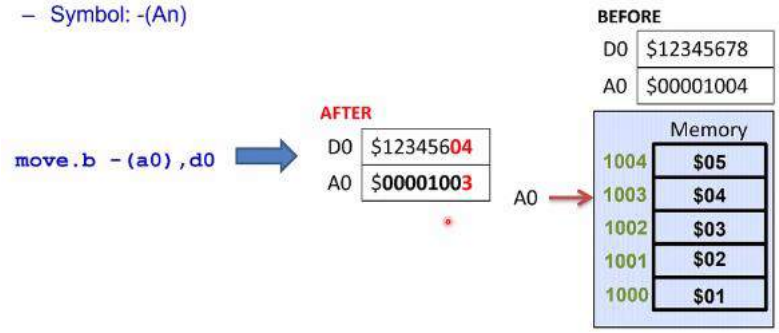




Pre-Decrement Indirect Addressing

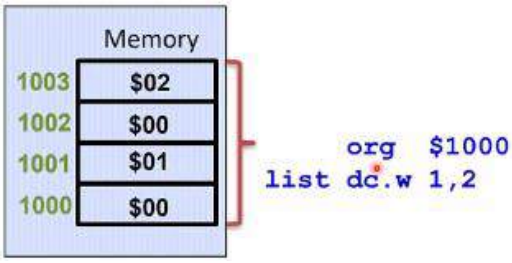
- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used

– Symbol: -(An)



Post Increment/Pre Decrement and C

- Consider the example
  - ptr = &list;
  - sum += *ptr++;



- Assume that the list variables are words (i.e., shorts) and sum is contained in D0 and the pointer ptr is to be placed in A0

```
lea      list,a0     ;a0=$00001000
add.w    (a0)+,d0    ;d0=d0+word at 1000 and a0=a0+2
```

LEA **Load Effective Address** **LEA**

Operation: < ea > → An

Assembler Syntax: LEA < ea > ,An

Attributes: Size = (Long)

Description: Loads the effective address into the specified address register. All 32 bits of the address register are affected by this instruction.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	REGISTER			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

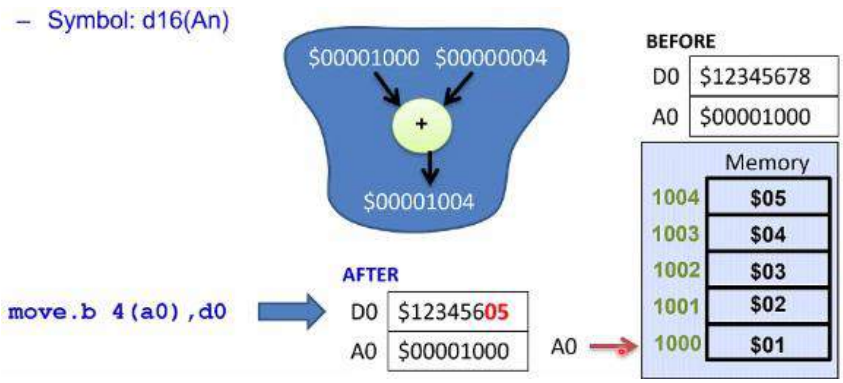
Register field—Specifies the address register to be updated with the effective address.

Effective Address field—Specifies the address to be loaded into the address register. Only control addressing modes can be used as listed in the following table:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An) +	—	—			
− (An)	—	—			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011

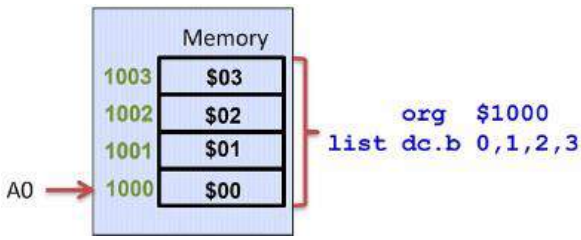
Indirect Addressing With Displacement

- Address of operand is contained inside an address register plus a sign-extended 16-bit displacement



Register Indirect with Displacement and C

- Consider the example
  - a = a + list[2];



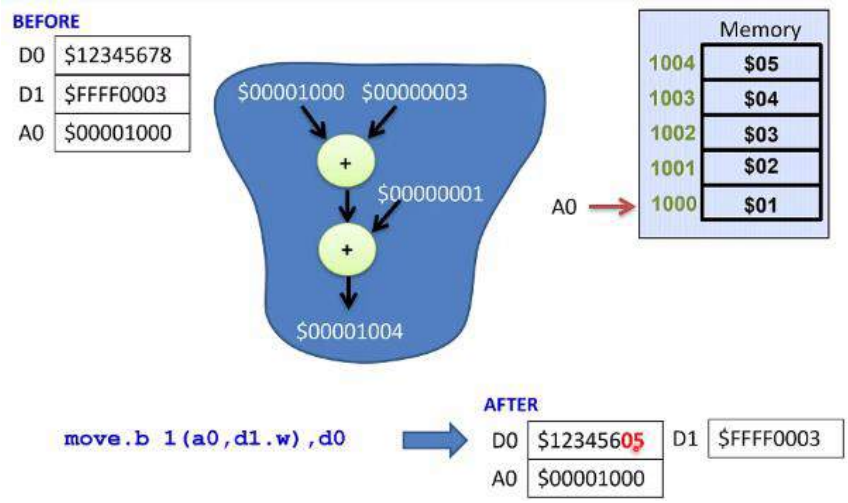
- Assume that the list variables are bytes (i.e., chars) and a is contained in D0 and the array (pointer) list is placed in A0

```
add.b   2(a0),d0   ;d0 = d0 + byte at 1002
```

```
00008000  41F9 00009000          10  START:  LEA     ARRAY,A0
00008006  1028 0000              11          MOVE.B  BEGIN(A0),D0
0000800A  1228 0004              12          MOVE.B  END(A0),D1
0000800E  1428 0002              13          MOVE.B  MIDDLE(A0),D2
00008012  1628 0000              14          MOVE.B  MIDDLE-2(A0),D3
00008016                         15
00008016                         16
00008016                         17
00009000                         18           ORG    $9000
00009000= 01 02 03 04 05         19  ARRAY    DC.B    1,2,3,4,5
00009005  =00000000              20  BEGIN    EQU     0              ; ARRAY[0]
00009005  =00000002              21  MIDDLE   EQU     2              ; ARRAY[2]
00009005  =00000004              22  END      EQU     4              ; ARRAY[4]
```

Indirect Addressing with Index and Displacement

- The address of operand is contained inside an address register plus an index register plus a sign-extended 8-bit displacement
  - Symbol: d8(An,Xn)
- Index register (Xn) can be either an address register or data register
  - Size
    - Xn.L
    - Xn.W (which will be sign-extended to 32 bits)

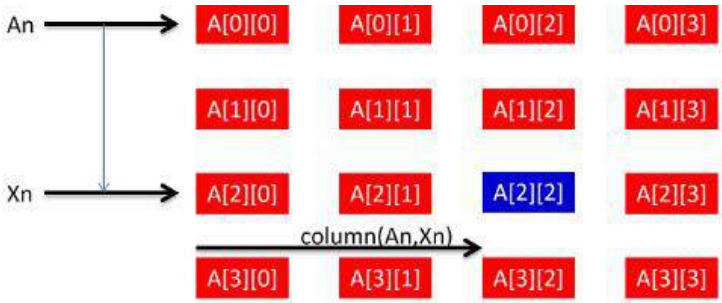


Register Indirect with Index plus Displacement and C

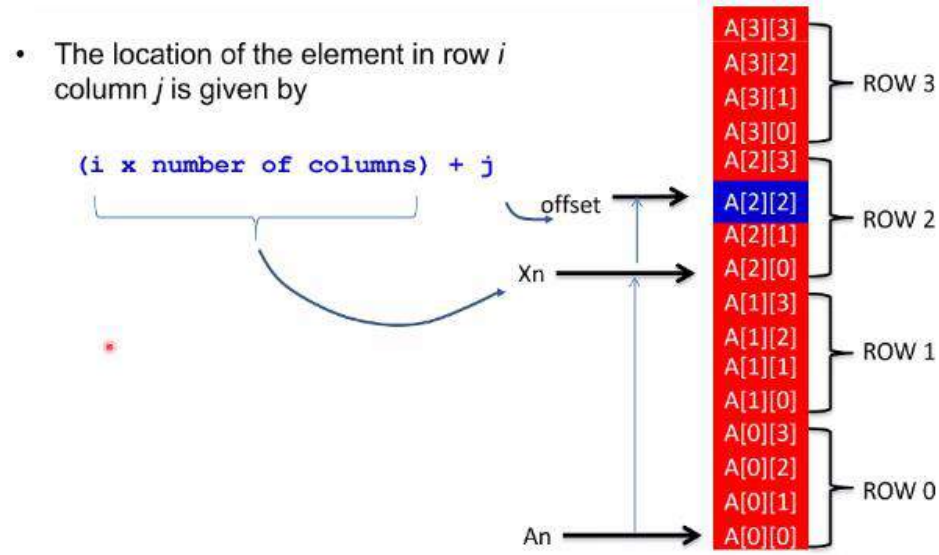
- Consider the example
  - a = a + list[i];
- Assume that the list variables are long words (i.e., ints) and a is contained in D0, i is contained in D1 and the array (pointer) list is in A0

```
move.w   #4,d3              ;d3 = 4
mulu     d1,d3              ;d3 = 4 * i
add.l    0(a0,d3.l),d0      ;d0 = d0 + list[i]
```

Accessing a 2D Array



C Stores Array in Row-Major Order



Assembler Code

- Consider the example
  - char list[4][4];
- Uninitialized array

```
ncols    equ      4
nrows    equ      4
list     ds.b     ncols*nrows
```

- Initialized array

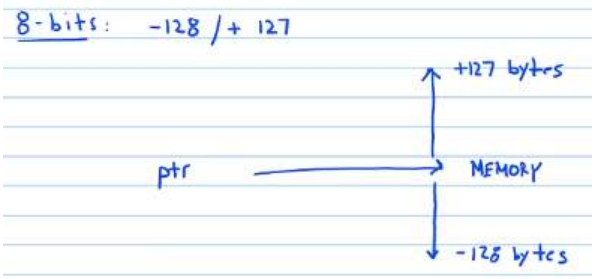
```
list     dc.b     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

2-d array (list)

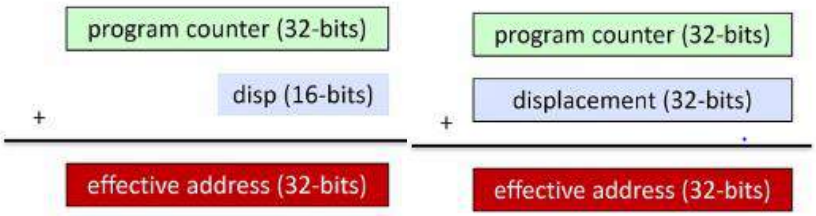
- Consider the example
  - a = a + list[2][1]; j = 1, i = 2
- Assume that the list variables are bytes (i.e., chars) and a is contained in D0 and the array (pointer) list is in A0

```
move.w   #ncols,d2          ;ncols = 4
mulu     #2,d2              ;i x ncols
add.b    1(a0,d2.l),d0      ;a0 + (i x ncols)+ j
```



Program Counter with Displacement

- Address of operand is computed by adding a 16-bit signed displacement (in extension word) to the PC
  - Symbol: d16(PC)
    - Displacement is usually expressed using a label
    - The value in the PC when the addition takes place is the address of the extension word



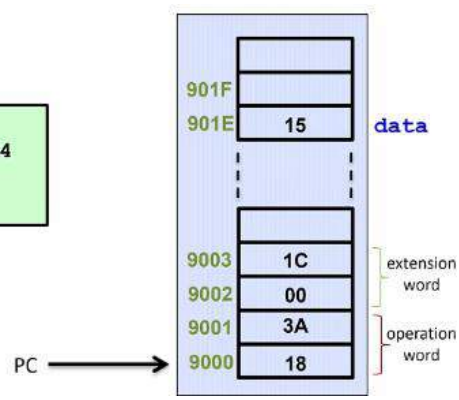
- Allows the program to load and execute at any address, no matter what the ORG value used during assembly

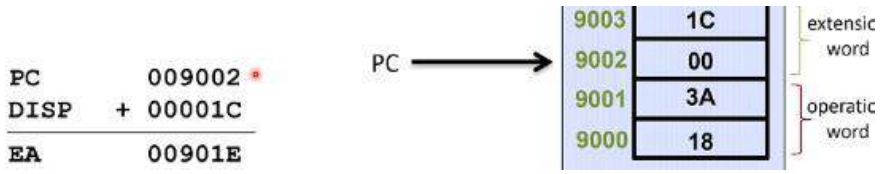
PC with Displacement - Processor Perspective

- Address of operand is computed by adding a 16-bit signed displacement (in extension word) to the PC
  - Symbol: d16(PC)

- Symbol: d16(PC)

```
        org     $9000
        move.b  data(pc),d4
        org     $091E
data    dc.b    $15
```





Address Modes with Instructions

- You can't assume you can use all address modes with all instructions, source and destinations
- Check in instructions

CIS2030 Microcomputers Week 5

Control Flow

- The default operation of the processor is to execute a straight-line sequence of instructions read from successive memory locations
- This results in two problems
    - Programs run until they reach the "end" of memory
    - Programs can't accomplish anything useful

Control Flow and Branch Instructions

- Branch instructions modify the flow of control and cause the program to continue execution at the target address specified by the branch
- Two types:
    - **Unconditional branch**
        - Like "goto" statement in C: goto L3;
        - Always forces a jump to the instruction at the target address (often specified using a label)
    - **Conditional branch**
        - Like "if-statement" in C: if(condition is true) goto L3;
        - Test condition and only branch to the target address (label) if condition is true

Branch Always (Unconditionally)

BRA Branch Always

Syntax: `BRA <target address>`

Name	Displacement	Machine Language	Operation Performed
BRA.S	8-bit (XX)	60XX	PC ← PC + displacement
BRA.L	16-bit (XXXX)	6000 XXXX	

```
        ORG     $1000
START   BRA     HERE
        NOP
        ORG     $1020
HERE    NOP
        BRA     HERE
        END     START
```

PC + disp = target address

disp = target address − PC

$= 1020_{16} - 1002_{16}$

$= 001E_{16}$ ✓

↳ byte

disp = target address − PC

$= 1020_{16} - 1024_{16}$

$= 7_{16}$ ✓ $(-4)_{10}$

Jump (Unconditionally)

JMP Jump

Syntax: `JMP <ea>`

Operation: PC ← destination address

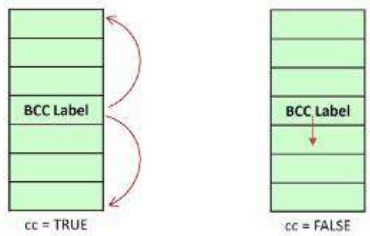
Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Xn)	ABS.W	ABS.L	Imm	d(PC)
		✓			✓	✓	✓	✓		✓

Conditional Branch Instructions

Bcc **Branch on condition cc**

Syntax: `Bcc <label>`
`Bcc <literal>`

Operation: if (cc = TRUE)
PC ← PC + displacement



Mnemonic	Flags
BEQ	Z=1
BNE	Z=0
BMI	N=1
BPL	N=0
BCS	C=1
BCC	C=0
BVS	V=1
BVC	V=0
BGE	Z=V
BGT	Z=0 and N=V
BLE	Z=1 or (N≠V)
BLT	N≠V
BHI	C=0 and Z=0
BLS	C=1 and Z=1

If-then Conditional Statement

- Consider the following C code:
  - if (a == b) goto Same;
- Assume that the variables are bytes (i.e., chars) and a is contained in D1 and b is contained in D0

```
sub.b d0,d1          ;Set or Clear Z flag
beq   Same;          ;goto Same if Z = 1
```

Instructions for Comparing Two Values

CMP **Compare**

Syntax: `CMP <ea>,<ea>`
Operation: CCR ← (destination – source)

Instruction	Source Operand	Destination Operand
CMP	Any address mode	Must be a data register
CMPA	Any address mode	Must be an address register
CMPI	An immediate value	Any address mode except address register indirect and immediate
CMPM	Compares one memory location with another. The only address mode permitted is address register indirect with post-incrementing	

If-then Solution

```
cmp.b d0,d1      ;compare values, discard result
beq   Same       ;goto Same if Z = 1
```

Conditional Branch Instructions Depending on a Single CCR Flag

- Branch instructions can be organized into pairs

Mnemonic	Condition	Flags
BEQ	equal	Z=1
BNE	not equal	Z=0
BMI	negative	N=1
BPL	positive or zero	N=0
BCS	carry set	C=1
BCC	carry clear	C=0
BVS	overflow set	V=1
BVC	overflow clear	V=0

Be Careful When Using BMI

Consider the following code

```
add.b d0,d1
bmi   $9000   ; branch taken if N=1 (sum is negative)
```

Before	After
D0 = 64_{10}	D0=64_{10}
D1 = 96_{10}	D1 = 160_{10}
CCR: X=0 N=0 Z=0 V=0 C=0	CCR: X=0 **N=1** Z=0 **V=1** C=0

Conditional Branch Instructions for Signed Numbers

mnemonic	cmp value1, value2	flags
BGE	value2 ≥ value1	N=V
BGT	value2 > value1	Z=0 and N=V
BLE	value2≤ value1	Z=1 or (N≠V)
BLT	value2 < value1	N≠V

- V-bit ensures that signed branches are correctly executed even when the compare operation produces overflow

Including Signed Overflow into Branches

Consider the following code

```
cmpi.b #-8,d0
bge    $9000  ; take branch if N=V (d0 >= -8)
```

Before	After
D0 = 127_{10}	D0 = 127_{10}
Calculation: $(127_{10} - (-8)_{10}) = 135_{10}$	
CCR: X=0 N=0 Z=0 V=0 C=0	CCR: X=0 **N=1** Z=0 **V=1** C=0

Conditional Branch Instructions for Unsigned Data

mnemonic	cmp value1, value2	flags
BHS , BCC	value2 ≥ value1	C=0
BHI	value2 > value1	C=0 and Z=0
BLS	value2≤ value1	C=1 and Z=1
BLO , BCS	value2 < value1	C=1

Signed and Unsigned Branches

Unsigned Comparison

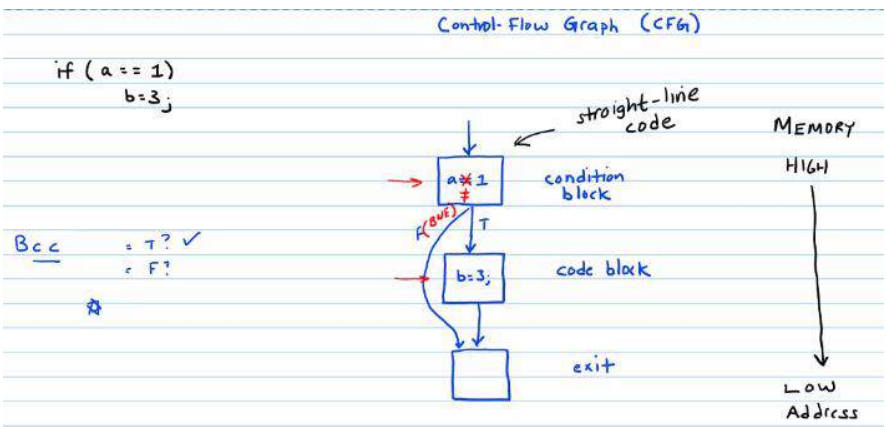
```
cmp.b d0,d1
bhs   $9000
```

- Signed Comparison

```
cmp.b d0,d1
bge   $9000
```

	Unsigned	Signed
D0 D1	00000010 (2) 11111111 (255)	00000010 (2) 11111111 (-1)
Result	11111101 (253)	11111101 (-3)
CCR	X=0, N=1, Z=0, V=0, **C=0**	X=0, **N=1**, Z=0, **V=0**, C=0
Branch	C=0	N=V

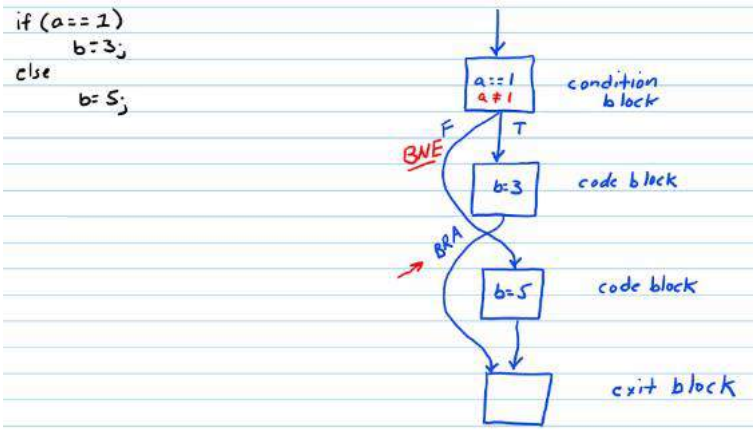
Simple “if” statement




```
cmpi.l   #1,a
bne      exit
```

condition

Simple “if-else” Statement



C code	Assembly Language

```
int a,b;
.
.
if (a == 1)
    b=3;
else
    b=5;
```

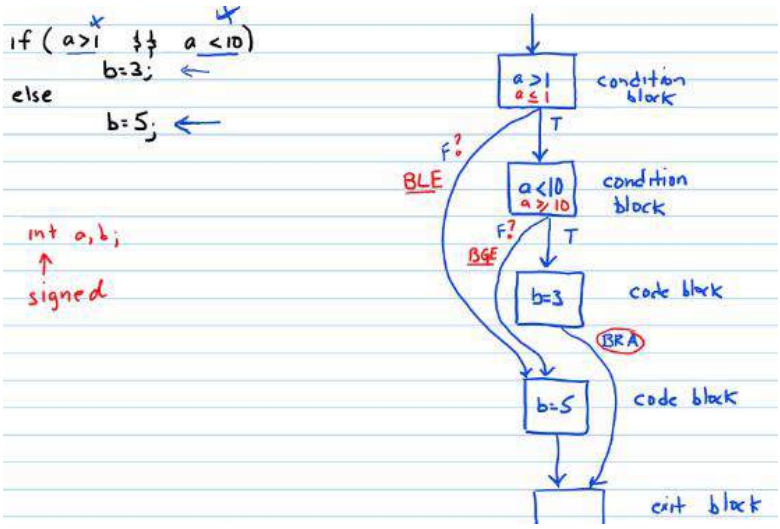
```
        org       $8000
        cmpi.l    #1,a
        bne       else
        move.l    #3,b
        bra       exit
else    move.l    #5,b
exit    move.b    #9,d0
        trap      #15
        org       $9000
a       ds.l      1
b       ds.l      1
```

`cmpi.l #1,a` `bne else`	condition
`move.l #3,b` `bra exit`	code
`move.l #5,b`	code
`move.b #9,d0` `trap #15`	exit

Complex Condition

```
int a,b;
.
.
if(a>1 && a<10)
    b=3;
else
    b=5;
```

With ANDs and ORs C uses short-circuit evaluation, in which it stops evaluating the condition as soon as it finds that it must be true or false no matter what the rest of the evaluation would give.



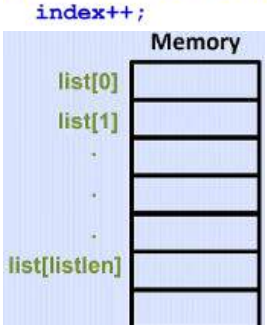

```
        org     $8000
        cmpi.l  #1,a
        ble     else
        cmpi.l  #10,a
        bge     else
        moveq   #3,b
        bra     exit
else    moveq   #5,b
exit    ...
        org     $9000
a       ds.l    1
b       ds.l    1
```

`cmpi.l #1,a` `ble else`	condition
`cmpi.l #10,a` `bge else`	condition
`moveq #3,b` `bra exit`	code
`moveq #5,b`	code
`...`	exit

Short-Circuit Evaluation

- Short circuit evaluation helps to
  - Improve code performance
  - Avoid runtime errors
- Consider the following code

```
index = 0;
while((index < listlen) && (list[index] != item))
  index++;
```



Switch Statements

```
typedef enum {ADD, MULT, MINUS, DIV,
   MOD, BAD}  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Implementation Options

1. Series of conditional branches
   - Good if few cases
   - Slow if many
2. Jump Table
   - Lookup branch target
     - Use 68000 **jmp** instruction to unconditionally jump to address stored in a register
   - Avoids conditional branches
   - Possible when cases are small integer constants

- C compiler e.g., gcc
  - Picks one based on case structure

Jump Table Structure

Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    . . .
  case val_n-1:
    Block n–1
}
```

Approximation Translation

```
target = JTab[op];
goto *target;
```

Jump Table

```
jtab:  Targ0
       Targ1
       Targ2
       .
       .
       .
       Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
         .
         .
         .
Targn-1: Code Block n–1
```

Switch Statement Example

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;
char unparse_symbol(op_type op){
  switch (op) {
    • • •
  }
}
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

- Assume op = D0, JumpTable = A1

```
cmpi.l   #0,d0          ;if op < 0
bls      exit           ;goto exit
cmpi.l   #6,d0          ;if op >= 6
bhs      exit           ;goto exit
mulu     #4,d0          ;compute displacement
movea.l  (a1,d0.l),a2   ;a2 = JumpTable[op]
jmp      (a2)           ;jump to JumpTable[op]
```

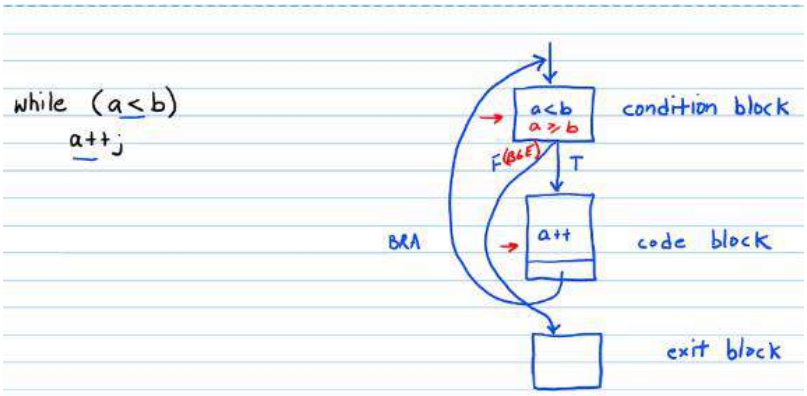
Jump Table

```
JumpTable
  .dc.l L0   ;Op = 0
  .dc.l L1   ;Op = 1
  .dc.l L2   ;Op = 2
  .dc.l L3   ;Op = 3
  .dc.l L4   ;Op = 4
  .dc.l L5   ;Op = 5
```

Targets

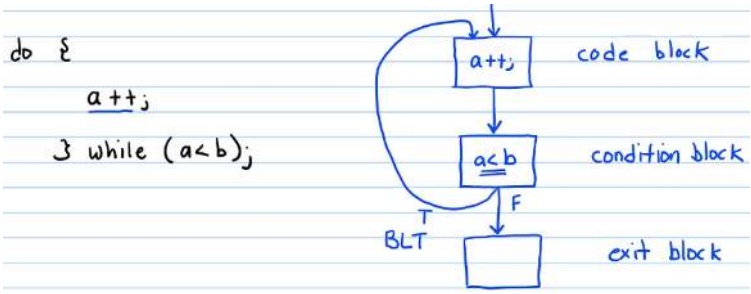
```
L0   move.b  #'+',d7
     jmp     exit

L1   move.b  #'*',d7
     jmp     exit

L2   move.b  #'-',d7
     jmp     exit

L3   move.b  #'/',d7
     jmp     exit

L4   move.b  #'%',d7
     jmp     exit

L5   move.b  #'?',d7

exit    ;end of switch
```

Simple “while” Statement



C code	Assembly Language			
`int a,b;`		`org`	`$8000`	
`.`		`move.l`	`a,d0`	condition
`.`	`top`	`cmp.l`	`b,d0`	
`while (a < b)`		`bge`	`exit`	
`a++;`		`addq`	`#1,d0`	code
		`bra`	`top`	
	`exit`	`...`		
		`org`	`$9000`	
	`a`	`ds.l`	`1`	
	`b`	`ds.l`	`1`	

“do-while” Statement



C code	Assembly Language			
`int a,b;`			`org`	`$8000`
`.`			`move.l`	`a,d0`
`.`		`top`	`addq`	`#1,d0` code
`do {`			`cmp.l`	`b,d0`
`a++;`			`blt`	`top` condition
`while (a < b);`		`exit`	`...`	
			`org`	`$9000`
		`a`	`ds.l`	`1`
		`b`	`ds.l`	`1`

While Loop

Consider the following C code

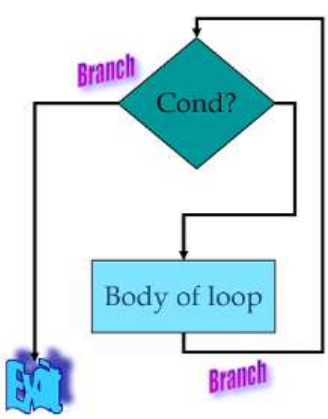
```
while (a[i] == k)
    i = i + j;
```

Assume `i`(D0), `j`(D1), `k`(D2) and `a`(A0)

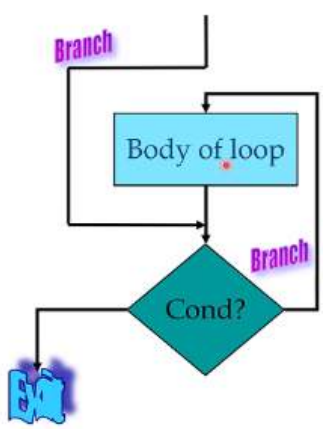
```
loop   cmp.b  (a0,d0.l),d2    ;a[i] == k?
       bne    exit            ;no
       add.l  d1,d0           ;i = i + j
       bra    loop            ;do it again
exit   ...
```

Improved While-Loop Efficiency

Code uses two branches/iteration:



- Better structure:



Improved Loop Solution

Remove extra branch from loop body

```
        bra     cond
loop    add.l   d1,d0           ;i = i + j
cond    cmp.b   (a0,d0.l),D2    ;a[i] == k?
        beq     loop            ;do it again
exit
```

- Reduced loop from 4 to 3 instructions
    - Even small improvements are important if loop executes too many times

For Statement

- Similar to while loop

```
init;
while (test)
        Body;
        update;
```

```
Init;
goto test;
loop:
   Body
   Update
test:
   if(test)
      goto loop;
```

```
for (init; test; update)
        Body;
```

```
Init;
goto test;
loop:
   Body
   Update
test:
   if(test)
      goto loop;
```

Original Loop

```
for i=0; a[i]==k; i=i+j
```

Optimized Loop

```
i=0 ;
  goto test ;
loop:
  i=i+j;
test:
  if a[i]==k goto loop ;
```

- Assembly Language

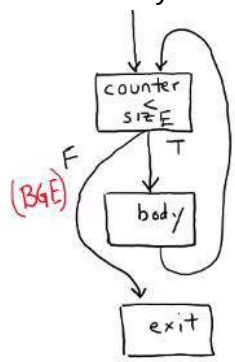
```
        move.l  #0,d0
        bra     test
loop    add.l   d1,d0
test    cmp.b   (a0,d0.l),d2
        beq     loop
exit
```

CIS2030 Microcomputers Week 6

Example 1 - Problem Description

- Write a program to sum an array of numbers
  - Assumptions
    - Array is stored at location LIST
    - Numbers are bytes
    - Label SIZE indicates the length of the array
    - Sum is to be stored at the end of the array

```
A0 ← ptr = &LIST;
     sum = 0; → D2
     counter = 0; → D1
                                  cmpi
                 register   label
     while(counter < SIZE){
        sum += *ptr++;
                 → (A0)++
        counter++;
     }
     *ptr = sum;
      → (A0)
```




```
* A0(pointer to LIST), D1 (loop counter) D2(sum)

        movea.l     #list,a0      ;initialize ptr
        clr.l       d2            ;sum=0
        clr.l       d1            ;counter=0
loop    cmpi.b      #size,d1      ;last byte?
        bge         exit          ;yes, goto exit
        add.b       (a0)+,d2      ;add to sum
        addq        #1,d1         ;counter++
        bra         loop          ;do it again
exit    move.b      d2,(a0)       ;store sum

        org         $9000
list    dc.b        1,2,3         ;array of bytes
sum     ds.b        1             ;location of sum
size    equ         sum - list    ;array size
```

MEMORY

	$9004
	$9003
$03	$9002
$02	$9001
$01	$9000

Symbol Table

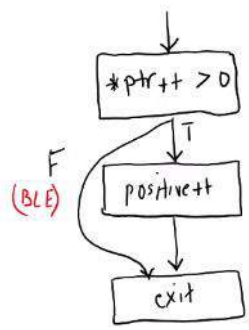
Symbol	Value
list	00009000
sum	00009003
size	00000003

Example 2 - Problem Description

- Write a program to count the number of (strictly) positive numbers in an array of numbers
  - Assumptions
    - Array is stored at location LIST
    - Numbers are bytes
    - Label SIZE indicates the length of the array
    - THe count is to be stored at the end of the array

```
ptr = &LIST;
positive = 0;
counter = 0;

while(counter < SIZE){
    if(*ptr++ > 0)
        positive++;
    counter++;
}
*ptr = positive;
```

(A0)+ constant cmpi




```
* A0(pointer to LIST), D1 (loop counter), D2(positive)
        movea.l     #list,a0      ;initialize ptr
        clr.b       d2            ;positive=0
        clr.b       d1            ;counter=0
loop    cmpi.b      #size,d1      ;last byte?
        bge         exit          ;yes, goto exit
        cmpi.b      #0,(a0)+      ;if byte not positive
        ble         cont          ;then don't count
        addq        #1,d2         ;add to positive
cont    addq        #1,d1         ;counter++
        bra         loop          ;do it again
exit    move.b      d2,(a0)       ;store positive
                                  ;return to 68KMB
        org         $9000
list    dc.b        1,-2,-3       ;array of bytes
pos     ds.b        1             ;location of positive
size    equ         pos - list    ;array size
```

Testing a Single Operand

TST **Test an Operand**

Syntax: `TST <ea>`

Operation: CCR ← operand - zero

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Xn)	ABS.W	ABS.L	Imm
✓		✓	✓	✓	✓	✓	✓	✓	

Example 2 - Revisited

```
* A0(pointer to LIST), D1 (loop counter), D2(positive)
        movea.l     #list,a0      ;initialize ptr
        move.b      d2            ;positive=0
        move.b      d1            ;counter=0
loop    cmpi.b      #size,d1      ;last byte?
        bge         exit          ;yes, goto exit
        tst.b       (a0)+         ;byte negative or zero?
        ble         cont          ;then don't count
        addq        #1,d2         ;add to positive
cont    addq        #1,d1         ;counter++
        bra         loop          ;do it again
exit    move.b      d2,(a0)       ;store sum
                                  ;return to 68KMB
        org         $9000
list    dc.b        1,-2,-3       ;array of bytes
pos     ds.b        1             ;location of positive
size    equ         pos - list    ;array size
```

Example 3

- How would you modify the previous program if the numbers in the array were unsigned and you wanted to find those values greater than 127?

```
ptr = &LIST;
counter = 0;
greater = 0;

while(counter < SIZE){
   if(*ptr++ > 127)
      greater++;
   counter++;
}
*ptr = greater;
```

Reverse branch, but ≤ → BLS

→ cmpi

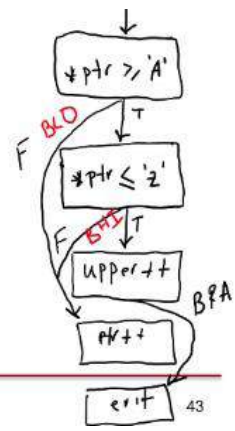
```
* A0(pointer to LIST), D1 (loop counter) D2(greater)
        movea.l     #list,a0      ;initialize ptr
        move.b      d2            ;positive=0
        move.b      d1            ;counter=0
loop    cmpi.b      #size,d1      ;last byte?
        bge         exit          ;yes, goto exit
        cmpi.b      #127,(A0)+    ;byte <= 127?
        bls         cont          ;then don't count
        addq        #1,d2         ;add to greater
cont    addq        #1,d1         ;counter++
        bra         loop          ;do it again
exit    move.b      d2,(a0)       ;store sum
                                  ;return to 68KMB
        org         $9000
list    dc.b        1,-2,-3       ;array of bytes
grt     ds.b        1             ;location of greater
size    equ         grt - list    ;array size
```

Example 4

- How would you modify the previous program if the array contained characters and you wanted to count the number of upper-case alphabetical characters?

```
ptr = LIST;
counter = 0;
upper = 0;

while(counter < SIZE){
    if(*ptr >= 'A' && *ptr <= 'Z')
        upper++;
    ptr++
    counter++;
}
*ptr = upper;
```

(A0) cmpi (A0) cmpi ADDA



43

```
* A0(pointer to LIST), D1 (loop counter) D2(upper)
        movea.l     #list,a0      ;initialize ptr
        move.b      d2            ;positive=0
        move.b      d1            ;counter=0
loop    cmpi.b      #size,d1      ;last byte?
        bge         exit          ;yes, goto exit
        cmpi.b      #'A',(A0)     ;byte before 'A'?
        blo         cont          ;then don't count
        cmpi.b      #'Z',(A0)     ;byte after 'Z'?
        bhi         cont          ;then don't count
        addq        #1,d2         ;add to upper
cont    addq        #1,d1         ;counter++
        adda.l      #1,a0         ;point to next char
        bra         loop          ;do it again
exit    move.b      d2,(a0)       ;store sum
                                  ;return to 68KMB
* data does not change
```

Example 5 - Problem Description

- Write a program to reverse the characters in a character array that is terminated with a carriage return
  - Assumption
    - No additional memory is available

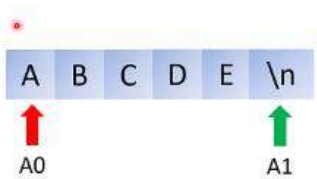
```
* A0/A1(ptrs to start/end of string), D0 (temporary)

* Position ptr (a1) at end of string
        movea.l     #string,a1     ;point to start of string
loop1   cmpi.b      #$D,(a1)+      ;carriage return?
        bne         loop1          ;no: try again
        suba.l      #1,a1          ;backup one byte
* Reverse characters
        movea.l     #string,a0     ;point to start of string
loop2   cmpa.l      a0,a1          ;if A1 <= A0
        bls         exit           ;then string is reversed
        move.b      (a0),d0        ;otherwise
        move.b      -(a1),(a0)+    ;swap
        move.b      d0,(a1)        ;bytes
        bra         loop2          ;do it again!
exit                               ;return to 68KMB
string  dc.b        'ABCDE',$D     ;string to reverse
```

Determining End Of String

```
* Position ptr (a1) at end of string
        movea.l     #string,a1     ;point to start of string
loop1   cmpi.b      #$D,(a1)+      ;carriage return?
        bne         loop1          ;no(Z=0): try again
        suba.l      #1,a1          ;backup one byte
```

Swapping Characters

```
* swap pair of chracters
        move.b      (a0),d0        ;save bottom char
        move.b      -(a1),(a0)+    ;move top char to bottom
        move.b      d0,(a1)        ;move bottom char to top
```



swap a & b

```
temp = a;
a = b;
b = temp;
```

When is the String Reversed?

```
* Loop exit condition
  loop2        cmpa.l  a0,a1       ;if A1 <= A0
               bls     exit        ;then string is reversed
```

Logical (Boolean) Instructions

- The 68000 supports four types of Boolean instructions

Instruction	Operation	Mnemonic	X N Z V C
AND	to data register to memory immediate	AND ea,Dn AND Dn,ea ANDI #k,ea	×✓✓0 0
EOR	to data register Immediate	EOR ea,Dn EORI #k,ea	×✓✓0 0
NOT		NOT ea	×✓✓0 0
OR	to data register to memory Immediate	OR ea,Dn OR Dn,ea ORI #k,ea	×✓✓0 0

Logical And

Consider the following C code

```
char a = 0xFF;
char b = 0x7E;
a = a & b;
```

Truth Table

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Assume that a(D1), b(D0)

```
and.b d0,d1
```

- Source is referred to as “mask”

Logical OR

Consider the following C code

```
char a = 0x80;
char b = 0x7E;
a = a | b;
```

Truth Table

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Assume that a(D1), b(D0)

```
or.b d0,d1
```

Logical EOR

Consider the following C code

```
char a = 0xFF;
char b = 0x36;
a = a ^ b;
```

Truth Table

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Assume that a(D1), b(D0)

```
eor.b d0,d1
```

Logical NOT

Consider the following C code

```
char a = 0x7E;
a = ~a;
```

Truth Table

A	not A
0	1
1	0

Assume that a(D1)

```
not.b d1
```

Example

- Write a program to set the parity bit included with an ASCII character (byte) to zero, then convert the ASCII character to lowercase
    - Assume the byte is contained in D0

(1) ANDI.B %01111111,D0

(2) ORI.B %00100000,D0

Shift Operations

- There are six instructions that shift an operand one or places to the left or to the right

Logical Operation	C Operation	Signed Data	Unsigned Data
shift left	<<	ASL	LSL
shift right	>>	ASR	LSR
circular shift left	N/A	ROL	
circular shift right	N/A	ROR	

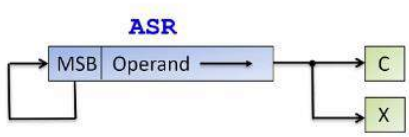
Expressing Shift Instructions in Assembly Language

- The shift count may be expressed in one of three ways

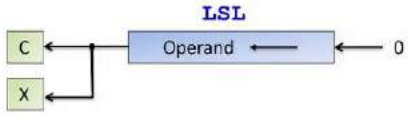
Shift Count	Example	Description
Immediate	ASL.W #3,D0	Allows a shift of 1-8 places
Register	ASL.L D0,D1	Allows a shift of 1-32 places
Value of 1	ASL (A0)	Shifts the WORD 1 place

Arithmetic Shift Instructions

V-bit in CCR is affected

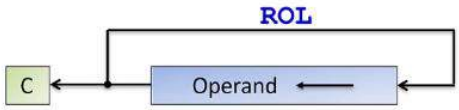


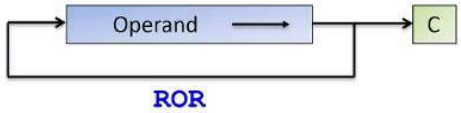
Logical Shift Instructions



V-bit in CCR is NOT affected

Circular Shift Instructions

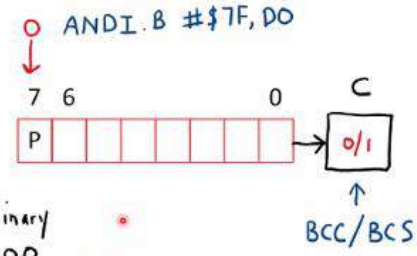




Example

- Write a program to set the parity bit (MSB) of a byte
- Assume that even parity is being used

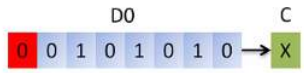
Algorithm
1. Set parity bit to 0
2. Count number of 1s in bit positions 0 through 6 — Rotate
3. If the number of 1s is odd, set parity bit to 1

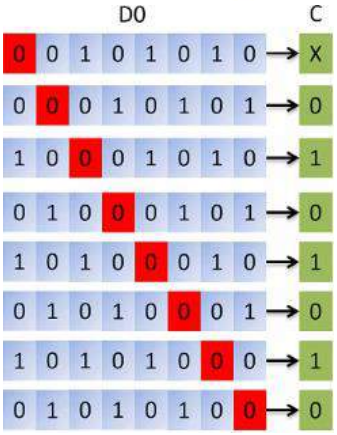


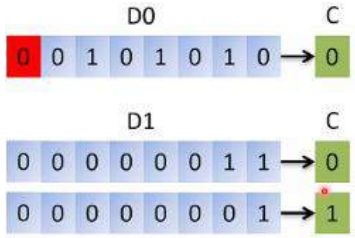
decimal	binary	
0	000	
1	001	← odd
2	010	
3	011	← odd
⋮	⋮	

```
        move.b      ascii,d0      ;original ascii character
        andi.b      #$7f,d0       ;clear parity bit
        clr.l       d1            ;clear parity counter
        move.l      #7,d2         ;loop counter
* Count number of bits equal to 1
loop    ror.b       #1,d0         ;rotate lsb into carry flag
        bcc         zero          ;bit equal to 1?
        addq        #1,d1         ;increment parity counter
zero    subq        #1,d2         ;repeat until all 7
        bne         loop          ;bits are tested
* Set parity bit if number of bits is odd
        ror.b       #1,d0         ;restore d0
        lsr.b       #1,d1         ;move lsb into carry
        bcc         exit          ;exit on even number
        ori.b       #$80,d0       ;set parity bit
exit    ...
ascii   dc.b        %00101010     ;original byte
```

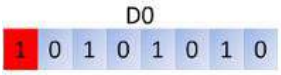
```
* Count number of bits equal to 1
loop    ror.b   #1,d0
        bcc     zero
        addq    #1,d1
zero    subq    #1,d2
        bne     loop
```







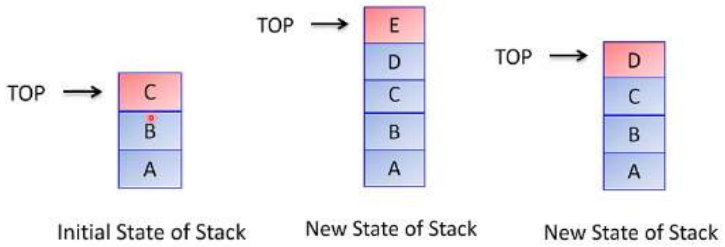

```
* Set parity bit if number of bits is odd
  ror.b     #1,d0
  lsr.b     #1,d1
  bcc       exit
  ori.b     #$80,d0
```



CIS2030 Microcomputers Week 7

Background - Stacks

- A stack is an ordered array of data items that can only be accessed at one end of the array, called the top
    - A push operation adds a new data item to the top of the stack
    - A pop operation removes the top item from the stack



Stack-Based Languages

- Stack-based languages, like C, make use of stacks for the following purposes
    - Call-return (pointer) mechanism
    - Arguments
        - Pass-by-value
        - Pass-by-reference
    - Local variables
    - Nested function calls
    - Recursive function calls
    - Reentrant code

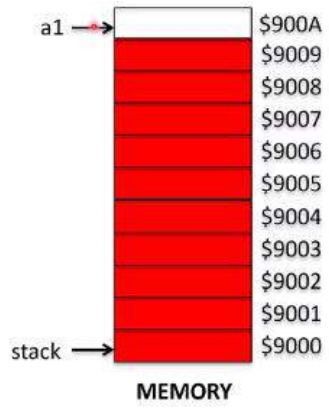
68000 Stack Conventions

- User Stacks on the 68000
    - Grow from the “top” of memory towards the “bottom” of memory
    - Employ an address register as a stack pointer to the top of the stack
        - System/User (runtime) starts
            - Two dedicated stack pointers: USP and SSP
            - Both implemented using A7
        - Other stacks
            - 7 general-purpose stack pointers: A0-A6
    - The 68000 does not provide push and pop instructions
        - **Push**: MOVE <ea>,-(An)
        - **Pop**: MOVE (An)+,<ea>

Creating a (Private) User Stack

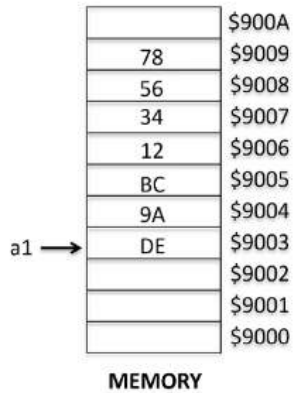
- Create a stack large enough to hold 10 bytes and use A1 as the stack pointer

```
        org     $9000
stack   ds.b    10
        lea     stack+10, a1
```

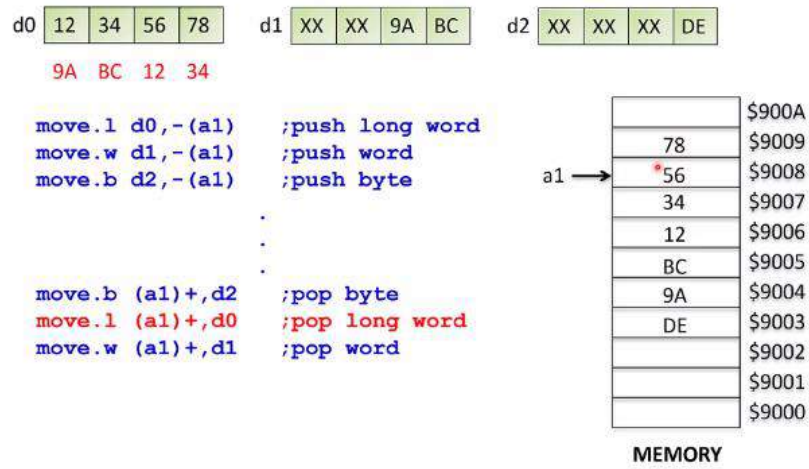


Examples of Push and Pop Operations

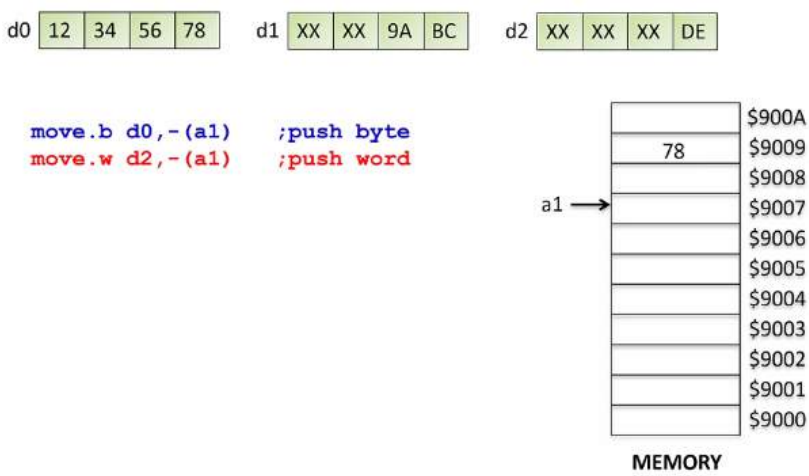
```
move.l d0,-(a1)    ;push long word
move.w d1,-(a1)    ;push word
move.b d2,-(a1)    ;push byte
                   .
                   .
                   .
move.b (a1)+,d2    ;pop byte
move.w (a1)+,d1    ;pop word
move.l (a1)+,d0    ;pop long word
```



Unbalanced Stack

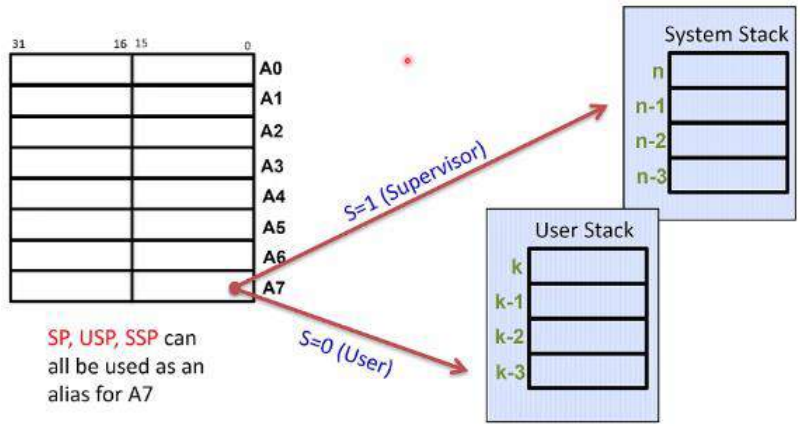


Address Error Exception



System and User Runtime Stacks and A7

- Address register A7 is used as a dedicated stack pointer

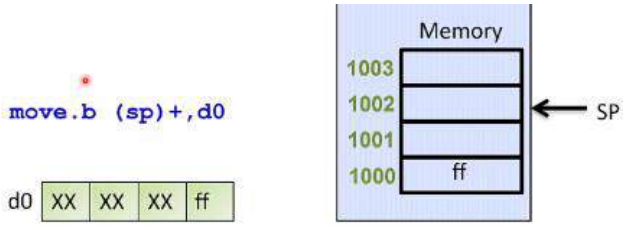


Common Stack Operations

Operation	Syntax	Description
Push to hardware stack	`move <ea>,-(sp)`	Decrement SP, then move <ea> to stack
Pop from hardware stack	`move (sp)+,<ea>`	Move information from stack to <ea>, then increment SP
Push Address to hardware stack	`pea <ea>`	<ea> is computed and pushed to stack
Move to stack	`move <ea>,d(sp)`	Move <ea> to SP + d without changing SP
Move from stack	`move d(sp),<ea>`	Move from SP + d to <ea> without changing SP

Runtime Stack and Byte Operations

- Word boundaries must be maintained on the hardware stack
  - If the stack pointer is A0-A6
    - The pointer is incremented or decremented by 1 when performing a byte operation
  - If the stack pointer is A7 (SP)
    - The pointer is incremented or decremented by 2 when performing a byte operation



Subroutines

- A subroutine is a piece of code that can be called by name from one or more locations in a program whenever its functionality is required

```
*main program
...
<call> subroutine
<next instruction>
...
<call> subroutine
<next instruction>
...
*end main program
```

```
*subroutine
<first instruction>
...
<return>
*end of subroutine
```

Call and Return Using an Address Register

```
caller:
        lea next,a1
        jmp subroutine
next    ...
```

```
callee:

subroutine  ...
            jmp (a1)
```

- Pros:
  - Simple, fast
- Cons:
  - Forces programmer to keep track of return address
  - Can't perform nested (or recursive) calls

Call and Return Using a Mailbox

```
caller:

mailbox  ds.l    1
         move.l  #next,mailbox
         jmp     subroutine
next     ...
```

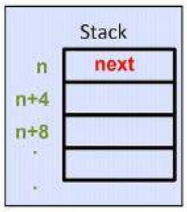
```
callee:

subroutine  ...
            movea.l mailbox,a1
            jmp (a1)
```

- Pros
  - Allowed for nested calls
- Cons

- Slow compared to register approach
- Can't perform recursive calls
- Memory space grows with number of functions (not with level of nesting)

Call and Return Using a Stack

```
caller:
        move.l #next,-(SP)
        jmp    subroutine
next    ...
```




```
callee:

subroutine  ...
            movea.l (sp)+,a1
            jmp (a1)
```

- Pros
  - Allows nested and recursive calls
  - Space grows with level of nesting
- Cons
  - Slow compared to register approach

Jump to Subroutine Instruction

JSR Jump to subroutine

Syntax: `JSR <ea>`

Operation: SP = SP – 4
Memory[SP] = Program Counter
Program Counter = <ea>

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Xn)	ABS.W	ABS.L	Imm
		✓			✓	✓	✓	✓	

Branch to Subroutine

BSR Branch to Subroutine

Syntax: BRS <label>
BRS <literal>

Name	Displacement	Machine Language	Operation Performed
BRS.S	8-bit	61XX	SP ← SP – 4 Memory[SP] = PC PC ← PC + displacement
BRS.L	16-bit	6100 XXXX	

Return from Subroutine Instruction

RTS Return from subroutine

Syntax: RTS

Operation: Program Counter = Memory[SP]
SP = SP + 4

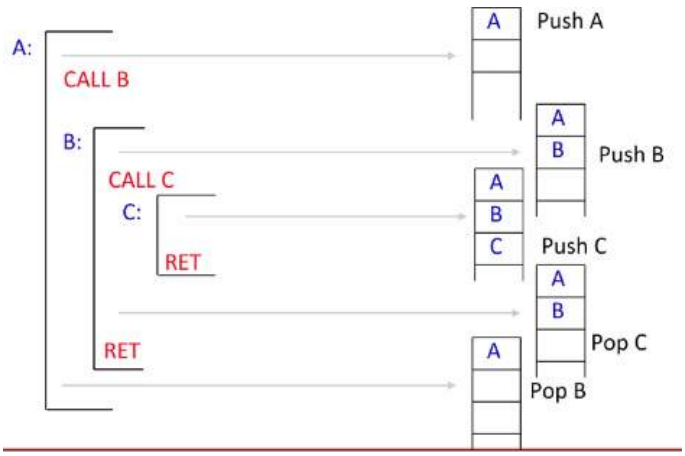
Example

- Write a subroutine to calculate $2X^2$
- Assumptions:
  - Subroutine is to be called SQR
  - X is a 16-bit signed value
  - X is to be passed to the subroutine using D0
  - $2X^2$ is to be returned to the subroutine using D0

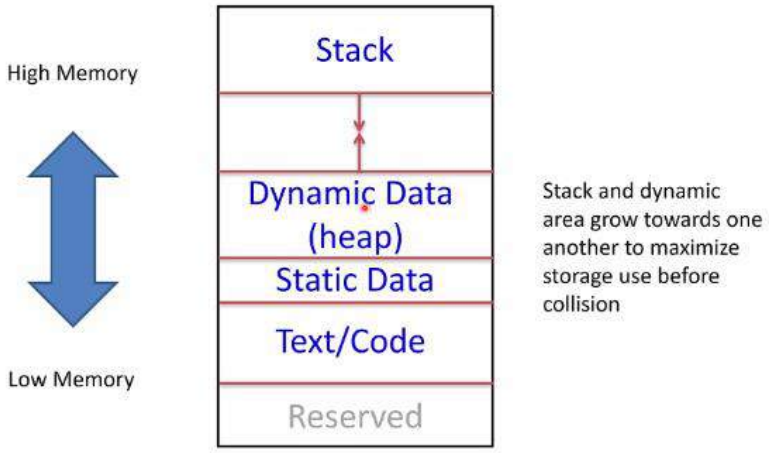
```
** Calling code

        MOVE.W  #4,D0           ; x = 4
        BSR     SQR             ; Call SQR(4)
        NOP                     ; do nothing on return

** Subroutine SQR

SQR     MULS    D0,D0           ; x * x
        ASL.L   #1,D0           ; 2 * (x * x)
        RTS
```

Nested Subroutine (Stack) Calls

- The stack grows downward and shrinks upward to accommodate nested subroutine calls



68000 Storage Layout (Software Convention)



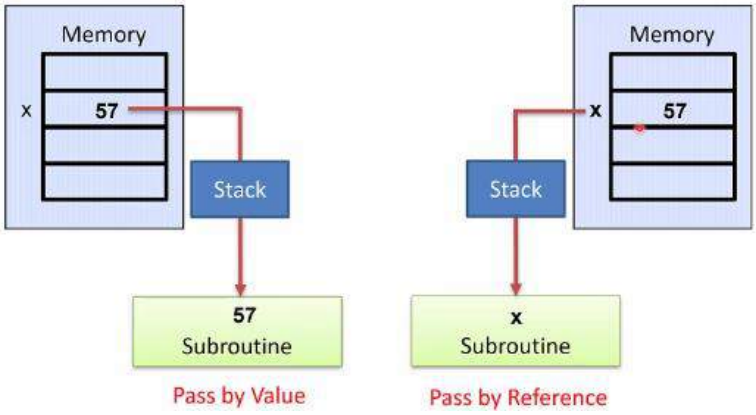
Implementing Functions

- Functions are required for structured programming
  - Aka: procedures, methods, subroutines
- Implementing functions in assembly requires several things to be done
  - Parameters must be passed in and return values passed out
    - Registers
    - Stack
  - Memory must be set aside for local variables
    - Heap
    - Stack
  - Execution must continue after the call
    - Stack

Passing Parameters on the Stack

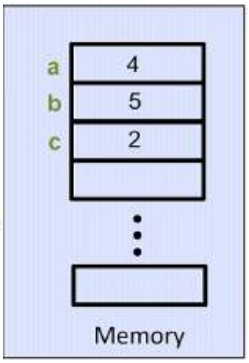
- Both the caller and the callee must know the number, order and type of parameters being passed
  - In C we use a function prototype
    - void Foo(int a, char b, int *c);
  - In assembler, you must take care of this yourself
  - C passes parameters right to left
- After returning from a subroutine
  - Calling code must remove any parameters from the stack
- Parameters may be passed by value or passed by reference
  - Pass-by-value: copy of parameter is passed
  - Pass-by-reference: address of parameter is passed

Pass by Value/Reference



C Function - Pass by Value

```
void foo() {
    short int a=4, b=5, c=2;
    register int product;

    product = mul3(a,b,c);
}


int mul3(short int x, short int y,
         short int z) {

    return (x*y*z);
}
```

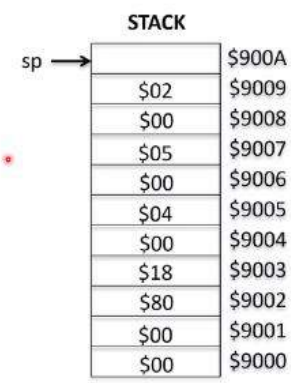


Assembly Code - Pass by Value

```
* foo() calling code
        move.w      c,-(sp)       ;push 1st parameter
        move.w      b,-(sp)       ;push 2nd parameter
        move.w      a,-(sp)       ;push 3rd parameter
        jsr         mul3          ;call subroutine
        lea         6(sp),sp      ;remove parameters


* mul3() multiplies 3 short ints and returns result in d0
mul3    move.w      4(sp),d0      ;d0 = a
        muls        6(sp),d0      ;d0 = a * b
        muls        8(sp),d0      ;d0 = a * b * c
        rts                       ;return

        org         $A000         ;variables
a       dc.w        4
b       dc.w        5
c       dc.w        2
```

Trace

```
******** foo() **********
8000            move.w    c,-(sp)
8006            move.w    b,-(sp)
800C            move.w    a,-(sp)
8012            jsr       mul3
8018            lea       6(sp),sp
801E

******** mul3() **********
8022     mul3   move.w    4(sp),d0
8026            muls      6(sp),d0
802A            muls      8(sp),d0
802E            rts

                org            $A000
A000     a      dc.w           4
A002     b      dc.w           5
A004     c      dc.w           2
```



PEA

PEA **Push Effective Address** **PEA**

Operation: SP – 4 → SP; < ea > → (SP)

Assembler Syntax: PEA < ea >

Attributes: Size = (Long)

Description: Computes the effective address and pushes it onto the stack. The effective address is a long address.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Field:

Effective Address field—Specifies the address to be pushed onto the stack. Only control addressing modes can be used as listed in the following table:

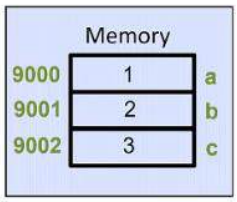
Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

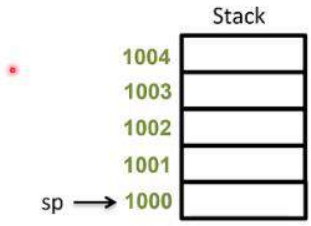
Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

PEA Instruction

Consider the following code

```
     org    $9000
a    dc.b   1
b    dc.b   2
c    dc.b   3
       .
       .
       .

     pea    a
```

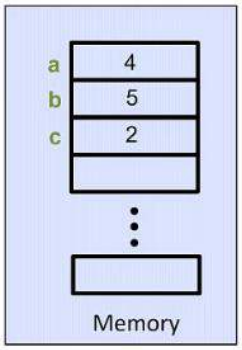




C Function - Pass by Reference

Consider the following code

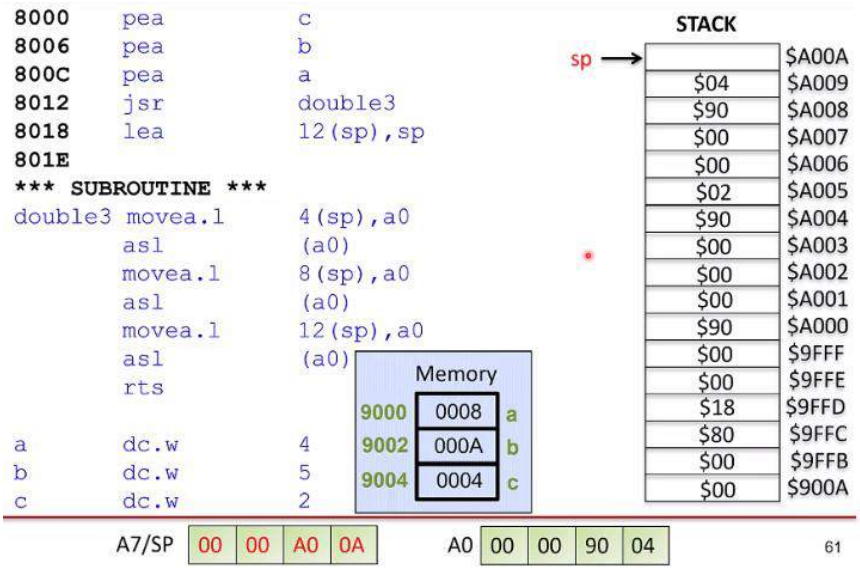
```
void foo() {
    short int a=4, b=5, c=2;
    double3(&a,&b,&c);
}


int double3(short int *x, short int *y,
            short int *z) {

    *x = *x * 2;
    *y = *y * 2;
    *z = *z * 2;
}
```



Assembler Code - Pass by Reference

```
        pea         c           ;push 3rd parameter
        pea         b           ;push 2nd parameter
        pea         a           ;push 1st parameter
        jsr         double3     ;call subroutine
        lea         12(sp),sp   ;remove parameters

***** double3() uses 3 pointers to double 3 ints *****
double3 movea.l     4(sp),a0    ;a0 = &x
        asl         (a0)        ;*x = *x * 2;
        movea.l     8(sp),a0    ;a0 = &y
        asl         (a0)        ;*y = *y * 2;
        movea.l     12(sp),a0   ;a0 = &z
        asl         (a0)        ;*z = *z * 2;
        rts                     ;return
        org         $9000       ;function parameters
a       dc.w        4
b       dc.w        5
c       dc.w        2
```

Trace



Transparency

- A transparent subroutine does not change any registers
  - Also known as “preserving registers across a call”
  - Achieving transparency
    - Register values are “saved” when first entering the subroutine
    - Restored prior to leaving the subroutine
- Where do we store the registers?
  - On the stack

MOVEM Instruction

MOVEM Move Multiple Registers

Syntax:
```
MOVEM reg.list,<ea>
MOVEM <ea>,reg.list
```

Operation: registers -> destination
source -> registers

- Pre-decrement address mode (Push)

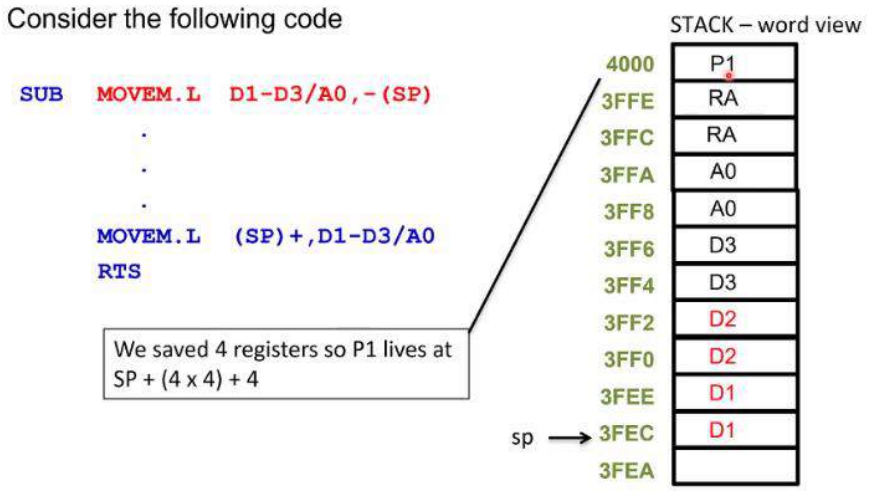
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

- Post-Increment address mode (Pop)

Order of transfer

A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

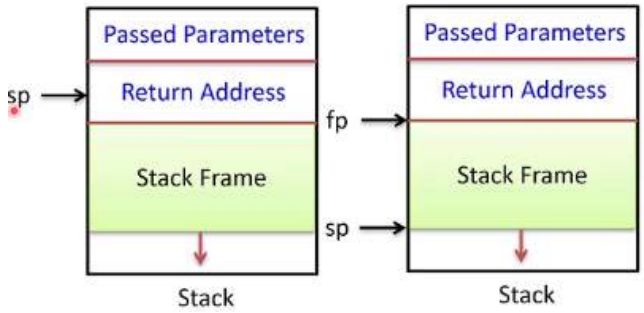
Example



CIS2030 Microcomputers Week 8

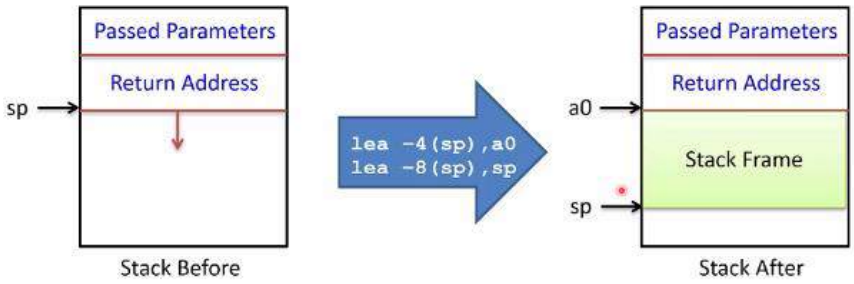
Local Variables and Stack Frames

- A subroutine frequently requires local variables
  - Local variables are temporary, intermediate values required only by the subroutine
- Two items closely associated with dynamic storage for local variables are
  - Stack frame
  - Frame pointer (fp)



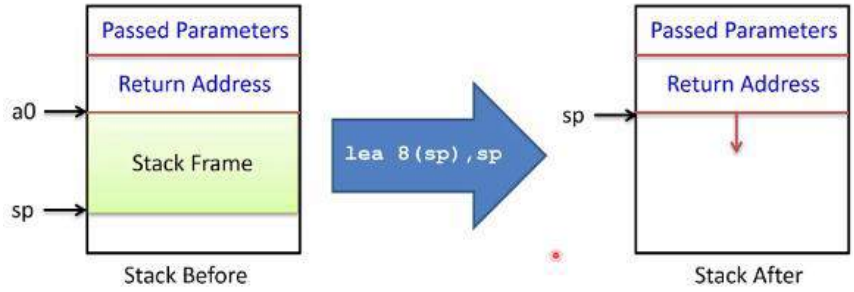
Manually Creating a Stack Frame

- Create a stack frame large enough to hold 8 bytes of local data and use A0 (point to first longword) as the frame pointer



Manually Destroy a Stack Frame

- Destroy the previous stack frame thus “freeing” the memory allocated to local variables

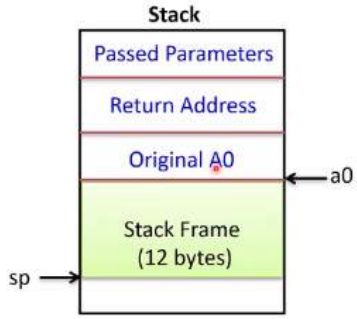


LINK Instruction

LINK Link and Allocate

Syntax: `link An,#-d`

Operation: SP = SP – 4
Memory[SP] = A0
A0 = SP
SP = SP - d



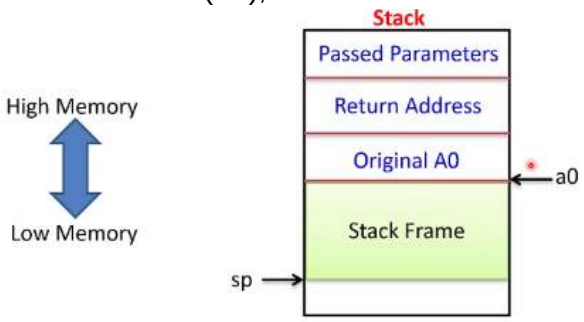
- Consider the following code

```
link a0,#-12
```

After Creating the Stack Frame

- Passed parameters are accessed with a positive displacement with respect to the frame pointer, A0
  - move +d(A0),Dn
- Local variables or temporary variables are accessed with a negative displacement with respect to the frame pointer, A0

- move -d(A0),Dn



UNLK Instruction

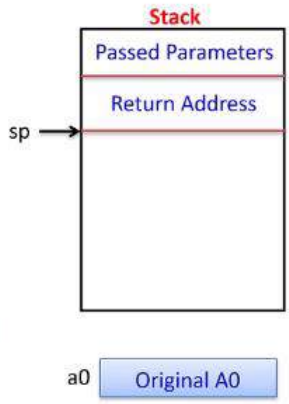
UNLK Unlink

Syntax: unlk An

Operation: SP = An
An=Memory[SP]
SP = SP + 4

- Consider the following code

```
unlk a0
```



Call and Return Conventions with Stack Frames

- Caller
  - Push parameters onto stack (MOVE Dn,-(sp) or PEA)
  - Call subroutine by executing JSR/BSR instruction
  - Remove parameters from stack upon return (LEA d(sp),sp)
- Callee Setup
  - Allocate memory for new frame (LINK An,#-d)
  - Save working registers (MOVEM reg.list,-(sp)), as needed
- Callee Running
  - Use frame pointer (An) and indirect addressing with offset (+-d(An)) to access parameters and local variables
- Callee Return
  - Put return value in data register (e.g., D0), if required
  - Restore working registers (MOVEM (SP)+,reg.lis)), as needed
  - De-allocate memory for stack frame (UNLK An)
  - Return by RTS

Using a Stack Frame for Intermediate Values

- Use a subroutine to calculate the following:
  - Result = $(P^2 + Q^2) / (P^2 - Q^2)$
- Assumptions
  - Parameters P and Q are 16-bit unsigned values passed to subroutine by value
  - The result is a 32-bit long word that is passed by reference
  - A minimum number of “working” registers must be used (i.e., 1 data register and 1 address register not including the frame pointer)
  - All working registers should be saved on the stack
  - A stack frame should be used to hold the temporary values P^2, Q^2, $P^2 + Q^2$, and $P^2 - Q^2$

Result : $(P^2 + Q^2) / (P^2 - Q^2)$

P - 16 bits
Q - 16 bits

P^2 - 32 bits (4 bytes)

Q^2 - 32 bits (4 bytes)

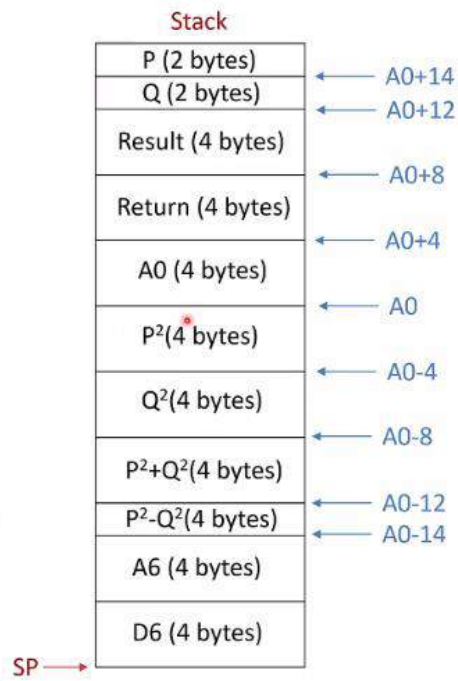
$P^2 + Q^2$ - dividend (4 bytes)

$P^2 - Q^2$ - divisor (2 bytes)

Total: 14 bytes

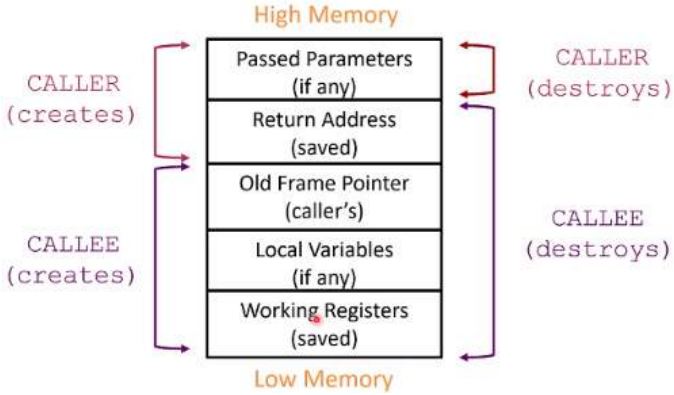
```
* Variables

P       DC.L    10              ; parameter 1
Q       DC.L    20              ; parameter 2
RESULT  DS.L    1               ; parameter 3


* Setup dummy values for working registers and fp

MAIN:
        LEA     $A6A6A6A6,A6    ; 1 working pointer
        LEA     $A0A0A0A0,A0    ; frame pointer
        MOVE.L  #$D6D6D6D6,D6   ; 1 working data register

* push parameters onto stack, call subroutine, and cleanup stack

        MOVE.W  P,-(SP)         ; pass P by value
        MOVE.W  Q,-(SP)         ; pass Q by value
        PEA     RESULT          ; pass Result by reference
        JSR     CALC            ; evaluate expression
        LEA     8(SP),SP        ; clean up!

        MOVE.B  #9,D0
        TRAP    #15             ; halt simulator
```

```
* subroutine CALC(P,Q,&RESULT)

CALC:
        LINK    A0,#-14          ; allocate 14-byte stackframe
        MOVEM.L D6/A6,-(SP)      ; save 2 working registers

        MOVE.W  14(A0),D6        ; get P from stack
        MULU    D6,D6            ; calculate P*P
        MOVE.W  D6,-4(A0)        ; save P*P on stackframe

        MOVE.W  12(A0),D6        ; get Q from stack
        MULU    D6,D6            ; calculate Q*Q
        MOVE.W  D6,-4(A0)        ; save Q*Q on stackframe

        MOVE.L  -4(A0),D6        ; get P*P from stackframe
        ADD.L   -8(A0),D6        ; calculate P*P + Q*Q
        MOVE.L  D6,-12(A0)       ; save P*P + Q*Q on stackframe

        MOVE.L  -4(A0),D6        ; get P*P from stackframe
        SUB.L   -8(A0),D6        ; calculate P*P - Q*Q
        MOVE.W  D6,-12(A0)       ; save P*P - Q*Q on stackframe

        MOVE.L  -12(A0),D6       ; get P*P + Q*Q from stackframe
        DIVU    -14(A0),D6       ; divide by P*P - Q*Q

        MOVEA.L 8(A0),A6         ; get address of RESULT
        MOVE.L  D6,(A6)          ; update value in address RESULT

        MOVEM.L (SP)+,D6/A6      ; restore working registers
        UNLK    A0               ; collapse stackframe
        RTS                      ; return to main


        END     MAIN             ; last line of source
```



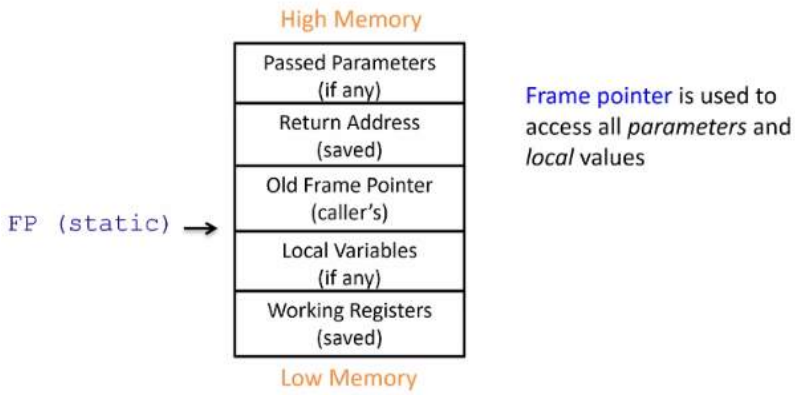
Activation Record

- Each call to a C function causes an activation record to be created on the runtime stack
  - Contains all data necessary for function to execute



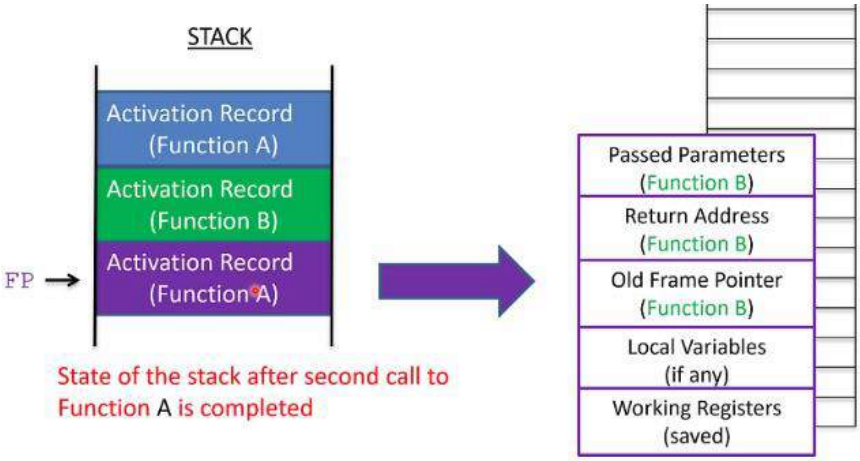
Activation Record - Frame Pointer

- The frame pointer associated with a particular activation record
  - Used by the function to access the contents of its activation record



Multiple Activation Records and the FP

- Multiple Activation Records build on top of one another, and the FP points to the active one



Example of Function Call

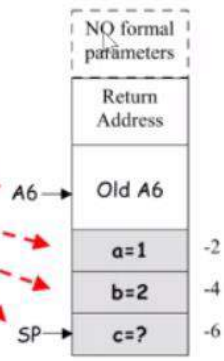
C Code	Assembler Produced by Compiler

```
int adder (int x, int y)
{
    return (x+y);
}


void main(void)
{
   int a,b,c;
   a=1; b=2;

   c=adder(a,b);
}
```

```
* int adder (int x, int y)

* Parameter x is at 8(A6)
* Parameter y is at 10(A6)
adder LINK    A6,#0
*       {
*     return (x+y);
      MOVE    8(A6),D0
      ADD     10(A6),D0
*       }
      UNLK    A6
      RTS
*

* void main(void)
* Variable a is at -2(A6)
* Variable b is at -4(A6)
* Variable c is at -6(A6)
main  LINK    A6,#-6
*       {
*     int a,b,c;
*     a=1; b=2;
      MOVE    #1,-2(A6)
      MOVE    #2,-4(A6)
*
*     c=adder(a,b);
      MOVE    -4(A6),-(A7)
      MOVE    -2(A6),-(A7)
      JSR     adder
      MOVE    D0,-6(A6)
      LEA     4(A7),A7
*       }
      UNLK    A6
      RTS
```

Assembler Code Produced by Compiler

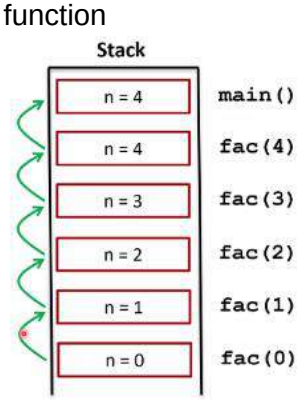
```
* int adder (int x, int y)

* Parameter x is at 8(A6)
* Parameter y is at 10(A6)
adder LINK    A6,#0
*       {
*     return (x+y);
      MOVE    8(A6),D0
      ADD     10(A6),D0
*       }
      UNLK    A6
      RTS
*

* void main(void)
* Variable a is at -2(A6)
* Variable b is at -4(A6)
* Variable c is at -6(A6)
main  LINK    A6,#-6
*       {
*     int a,b,c;
*     a=1; b=2;
      MOVE    #1,-2(A6)
      MOVE    #2,-4(A6)
*
*     c=adder(a,b);
      MOVE    -4(A6),-(A7)
      MOVE    -2(A6),-(A7)
      JSR     adder
      MOVE    D0,-6(A6)
      LEA     4(A7),A7
*       }
      UNLK    A6
      RTS
```



Recursion - Factorial

- A function is recursive if it is defined in terms of itself
  - n! = n x (n-1)! for n >= 1, 1 for n = 0
- Parameters must be bound to each individual use of the function
- Local variables must be bound to each individual use of the function
- A function must be able to return to the function that called it; return address must also be bound to each individual use of the function

```
int fac( 0 ) {

  if (FALSE)
      return n * fac(n-1);
  else
       return 1;
}


int main() {
    int n=4;
    printf("%d", fac(n));
    return 0;
}
```

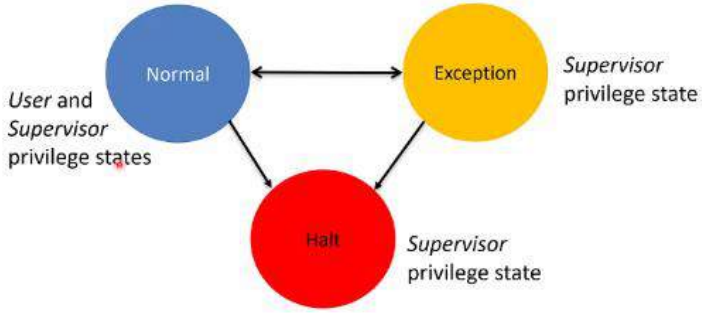


```
* Main function
MAIN:
        MOVE.W  #4,-(SP)        ; n=4
        BSR     FAC             ; compute fac(n)
        LEA     2(SP),SP        ; clean up stack


* Recursive function fac(n)
FAC     LINK    A6,#-2          ; n x fac(n-1) at -2(A6)

* if (n>=1)

        CMPI.W  #1,8(A6)
        BGE     YES             ; condition true?
* else
*    return 1

        MOVE.W  #1,D0           ; return 0! value in D0
        BRA     EXIT


* return n * fac(n-1)

YES     MOVE.W  8(A6),D1        ; get parameter n
        SUB     #1,D1           ; compute n-1
        MOVE.W  D1,-(SP)        ; push n-1
        BSR     FAC             ; compute fac(n-1)
        MULU    8(A6),D0        ; compute n x fac(n-1)
        MOVE.W  D0,-2(A6)       ; save n x fac(n-1)

EXIT    UNLK    A6
        RTS

        END     MAIN
```

CIS2030 Microcomputers Week 9

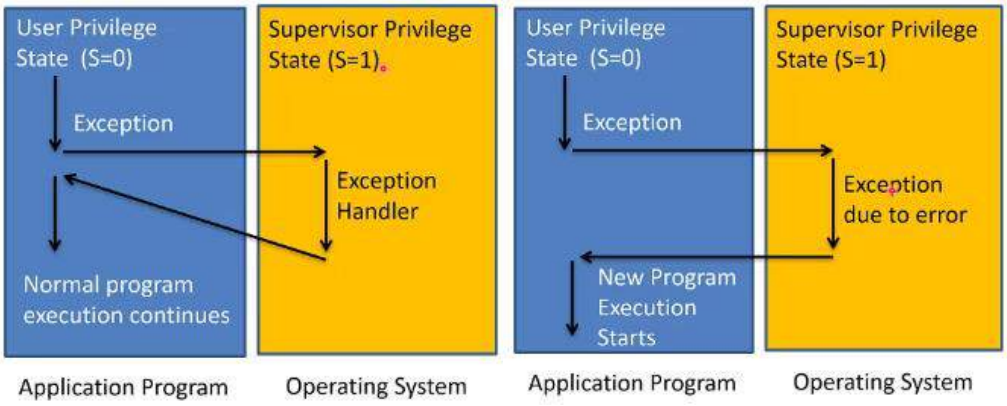
Execution/Privilege States

- The 68000 always in one of three processing states:
  - Normal
  - Halt
  - Exception



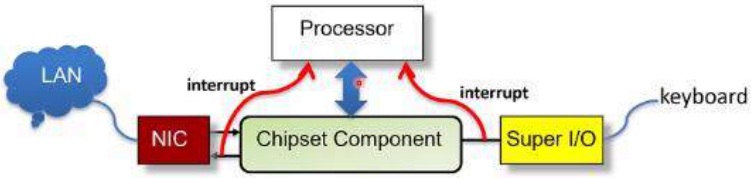
Exceptions in General

- Generated either in hardware or software, an exception causes the currently executing code to be effectively paused while the event is serviced by the operating system



Hardware Exceptions

- Hardware generated exceptions are
  - Called “interrupts”
  - Asynchronous events
  - Serviced only if external interrupt recognition is enabled (interrupt mask)

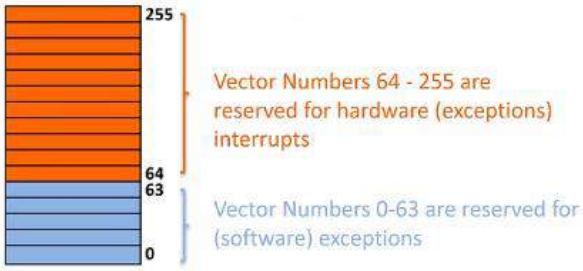


Software Exceptions

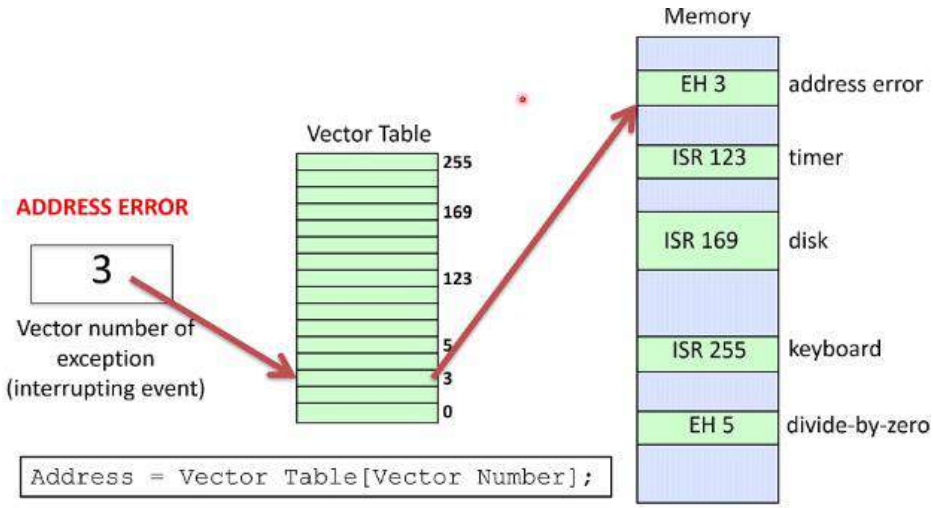
- Software generated exceptions are
  - Called “software interrupts” or just exceptions
  - Synchronous events
    - E.g., TRAP #N
  - Serviced regardless of the state of the interrupt mask
- Processor generated exceptions occur when the processor attempts to execute an instruction that requires additional attention
  - E.g., trying to divide by zero
  - Trying to access memory that is not present in RAM
  - Page fault (virtual memory)

Exception Cont'd

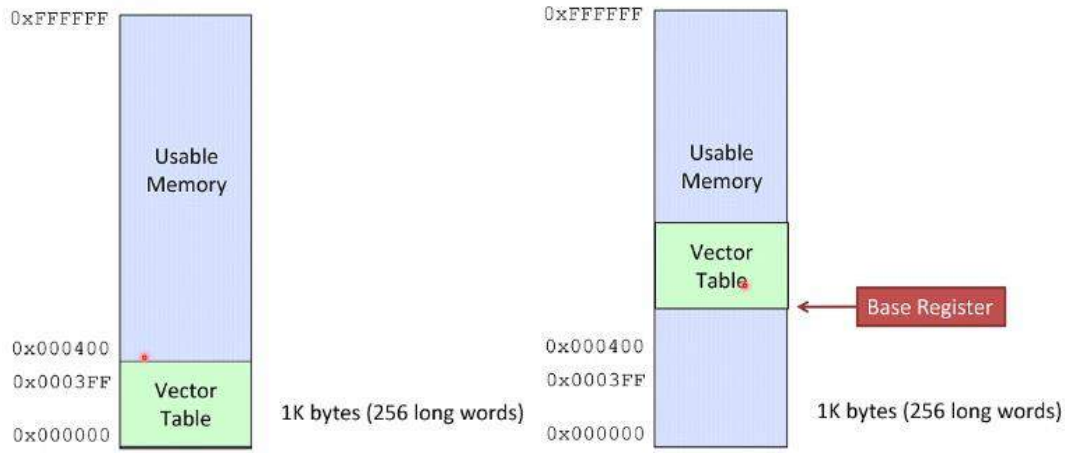
- Each exception is represented (identified) by a **Vector Number**
  - A vector number is an 8-bit (unsigned) value that identifies the exception and is defined by the ISA
- The vector number is used to locate the service routine to handle the specific event
  - Interrupt Service Routine (ISR) in the case of a hardware exception
  - Exception Handler (EH) in case of a software exception



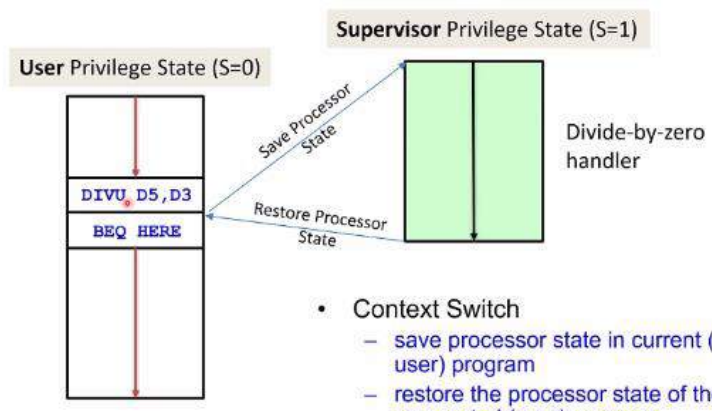
Finding the Exception Handler or ISR to Run



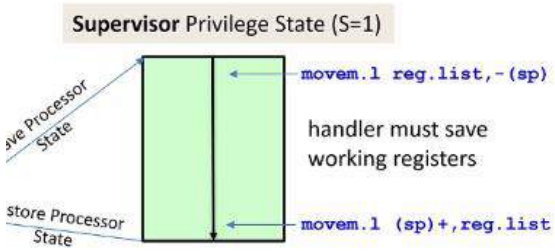
Location of Vector Table



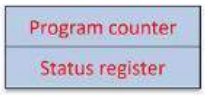
Context Switch



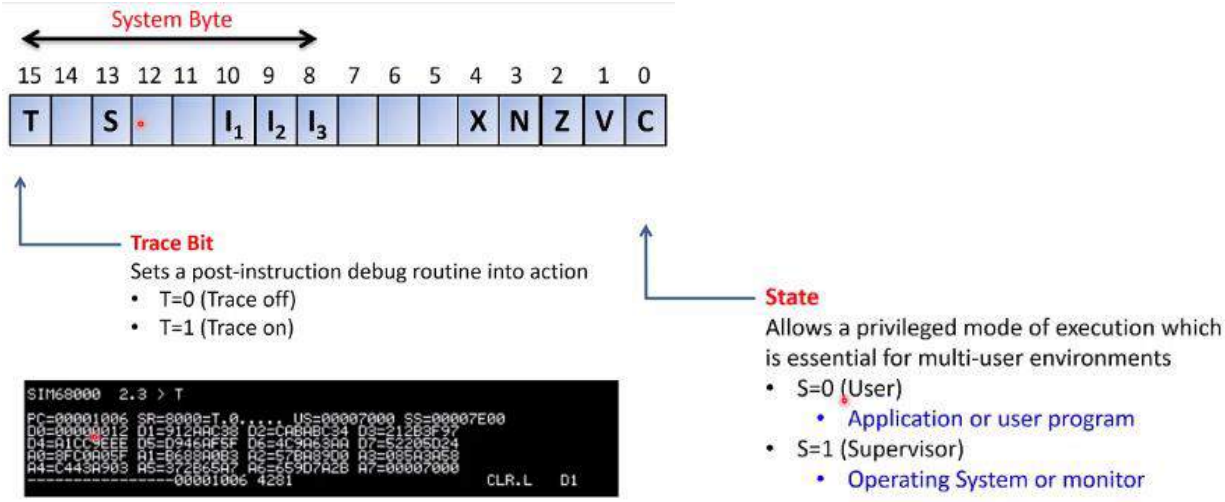
- Context Switch
  - save processor state in current (e.g., user) program
  - restore the processor state of the preempted (user) program
  - pass control to the restored (user) program
- Processor State
  - Program Counter
  - Condition-code Register flags
  - Privilege state
  - Working registers



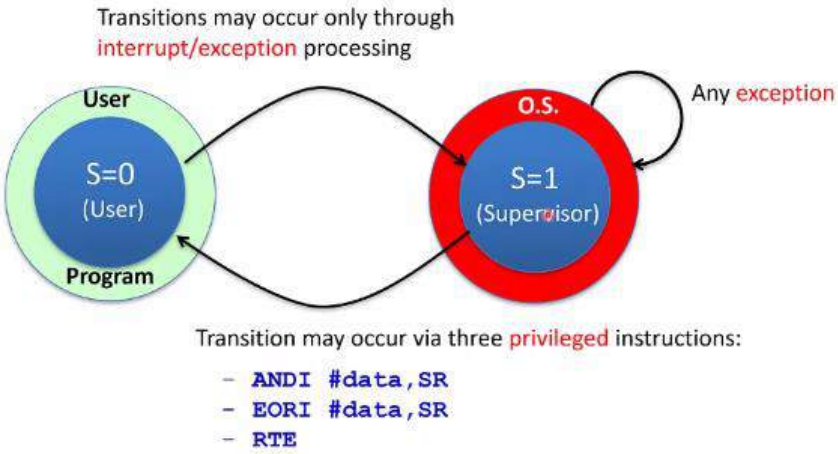
- Upon entry to the exception state, the processor sets S=1 and saves part of the context (processor state) on the stack



The Status Register and System Byte



Switch Between Supervisor and User States

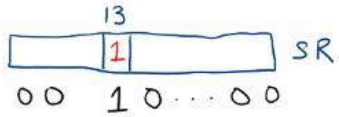


Difference Between Supervisor and User States

ISA Component	User (S=0)	Supervisor (S=1)
Active Stack Pointer	USP/A7	SSP/A7
Other Stacks	A0-A6	A0-A6, USP
Status Register Access	Read: Entire SR Write: CCR bits only	Read: Entire SR Write: Entire SR
Available Instructions	All Except: - ANDI #data,SR - EORI #data,SR - ORI #data,SR - MOVE <ea>,SR - MOVE USP,An - MOVE An,USP - RTE - RESET - STOP	All

Problem

- Show how the EORI instruction can be used to change the state of the processor from supervisor to user, without affecting any other bits in the status register

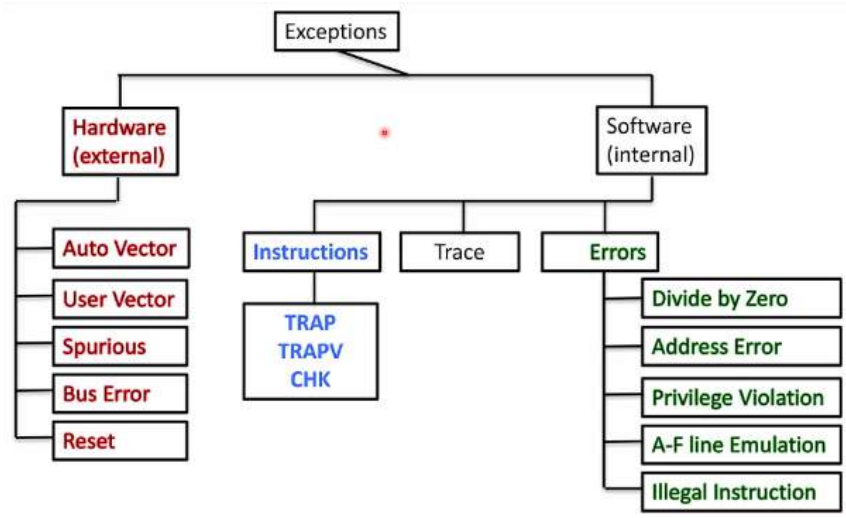


$0 \oplus n = n$

$1 \oplus n = \overline{n}$

EORI.W #$2000, SR

68000 Exception Taxonomy



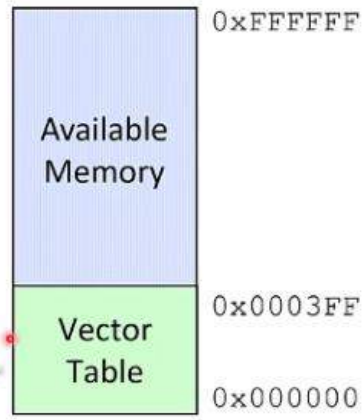
68000 Vector Table, Vector Numbers and Addresses

vector number (Decimal)	address (Hex)	assignment
0	0000	RESET: initial supervisor stack pointer (SSP)
1	0004	RESET: initial program counter (PC)
2	0008	bus error
3	000C	address error
4	0010	illegal instruction
5	0014	zero divide
6	0018	CHK instruction
7	001C	TRAPV instruction
8	0020	priviledge violation
9	0024	trace
10	0028	1010 instruction trap
11	002C	1111 instruction trap
12*	0030	not assigned, reserved by Motorola
13*	0034	not assigned, reserved by Motorola
14*	0038	not assigned, reserved by Motorola
15	003C	uninitialized interrupt vector
16-23*	0040-005F	not assigned, reserved by Motorola
24	0060	spurious interrupt
25	0064	Level 1 interrupt autovector
26	0068	Level 2 interrupt autovector
27	006C	Level 3 interrupt autovector
28	0070	Level 4 interrupt autovector
29	0074	Level 5 interrupt autovector
30	0078	Level 6 interrupt autovector
31	007C	Level 7 interrupt autovector
32-47	0080-00BF	TRAP instruction vectors**
48-63	00C0-00FF	not assigned, reserved by Motorola
64-255	0100-03FF	user interrupt vectors

NOTES:
* No peripheral devices should be assigned these numbers
** TRAP #N uses vector number 32+N

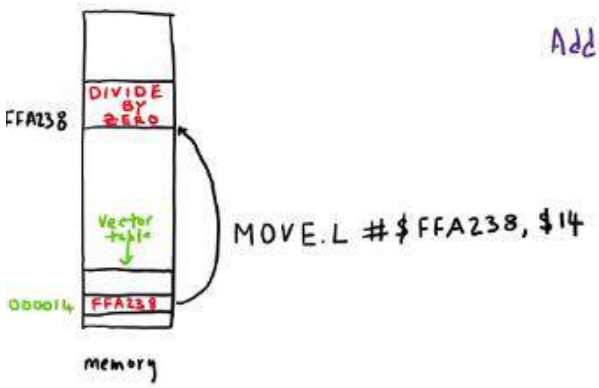
Vector Number
- Unique number associated with each exception

Vector address
- Address of the EH/ISR

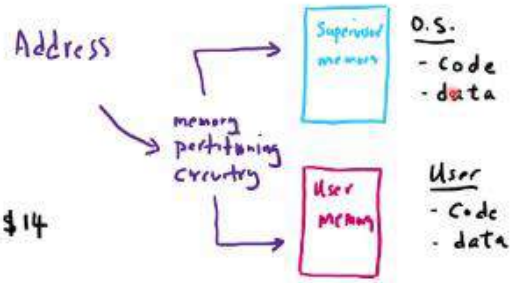


Problem

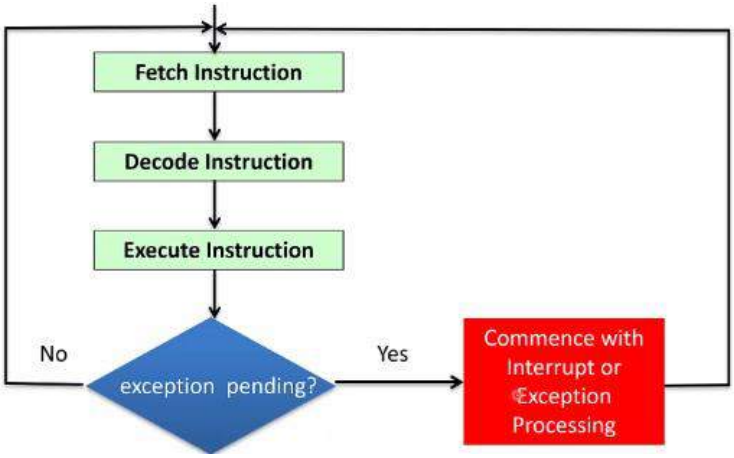
- A systems' programmer writes code for a divide-by-zero exception handler and places the routine at memory address $FFA238, write an instruction to update the vector table accordingly



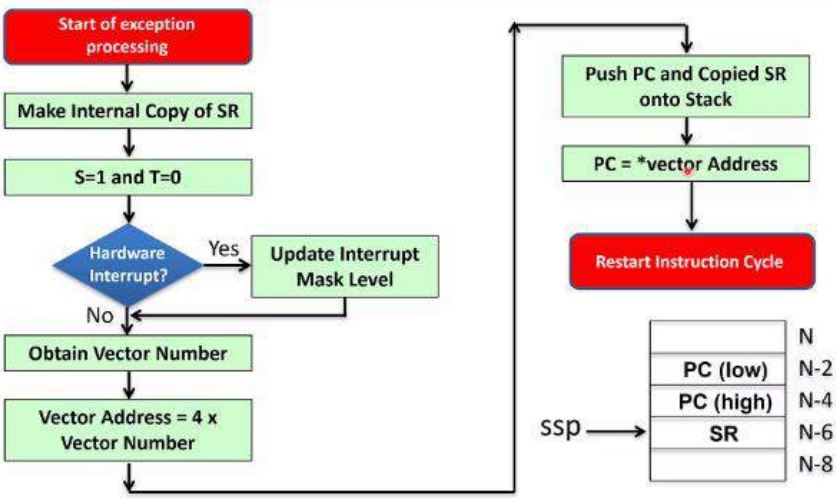
MOVE.L #$FFA238, $14



When are Exceptions Processed?



How are Exceptions Processed?

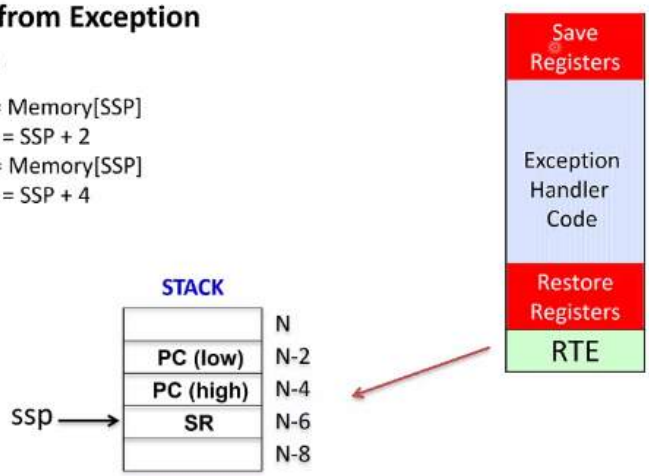


How to Return to Pending Program

RTE Return from Exception

Syntax: `RTE`

Operation:
SR = Memory[SSP]
SSP = SSP + 2
PC = Memory[SSP]
SSP = SSP + 4



TRAP Instruction

TRAP - used to perform operating system calls

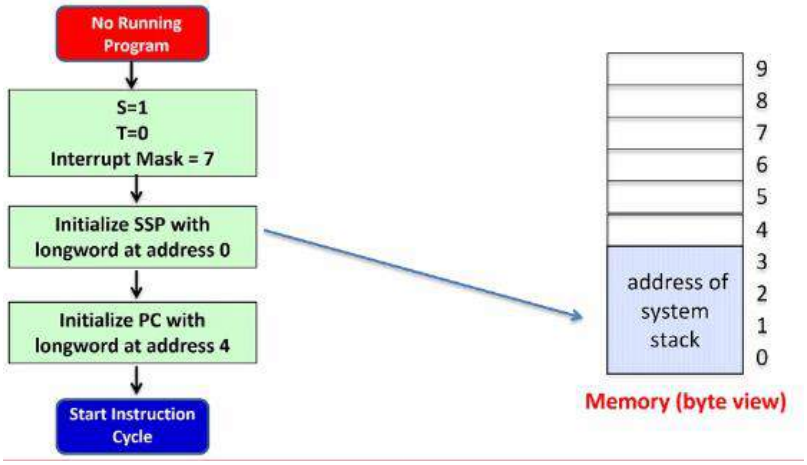
Syntax: `TRAP #N (0-15)`

Operation:
SSP = SSP - 4
Memory[SSP] = PC
SSP = SSP -2
Memory[ssp] = SR
PC = Vector Table [N + 32]

Note:
- possible to pass a integer to the trap handler in a data register
- Within the trap handler, the integer is used as an index into a jump table that points to the desired routine

Reset Exception

The reset exception occurs with the 68000 is first powered up



Exception Grouping and Priority

Highest

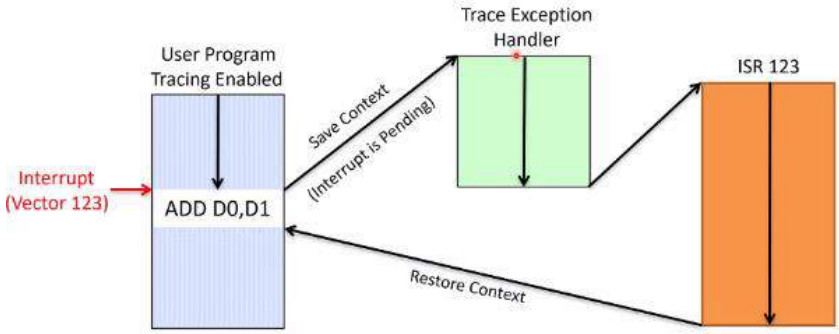
Priority

Lowest

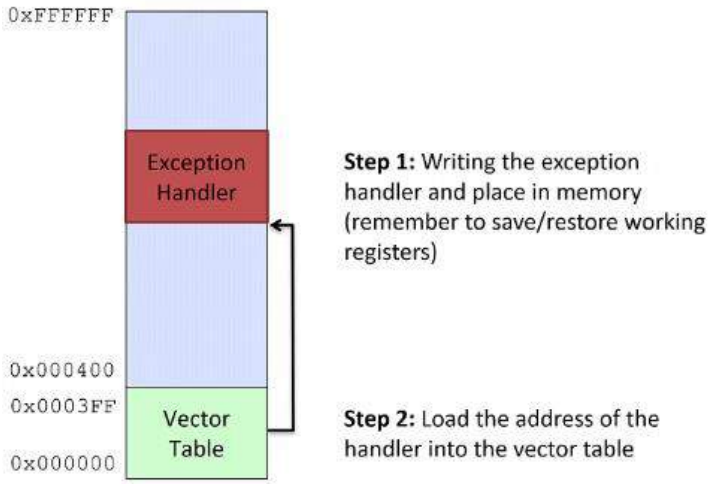
Group	Exception	Processing
0	Reset Address Error Bus Error	Processing begins within 2 clock cycles
1	Trace Interrupt Illegal Instruction Privilege Violation	Processing begins before the next instruction
2	TRAP, TRAPV, CHK Zero Divide	Processing is started by normal instruction execution

Problem

- What happens if the processor receives an interrupt request during the execution of a (user) instruction when tracing is on (i.e. T = 1)?



Writing Your Own TRAP Handler



Problem

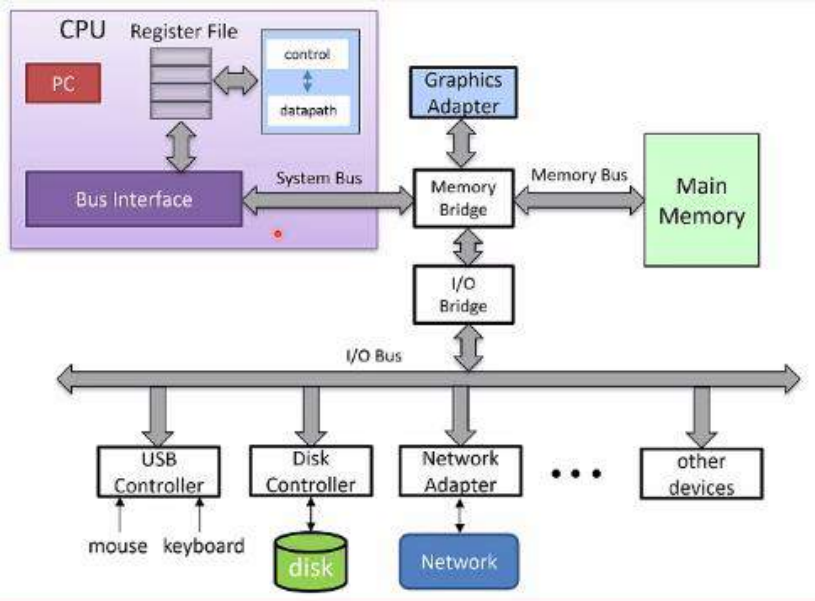
- Write a TRAP #5 exception handler to clear the bytes in memory between address 1 and address 2 inclusive
- Assume address 1 and 2 are passed to the handler in address registers A1 and A2, respectively, and address 1 is lower than address 2

Vector Number = $N + 32_{10}$
= $5_{10} + 32_{10}$
= 37_{10}

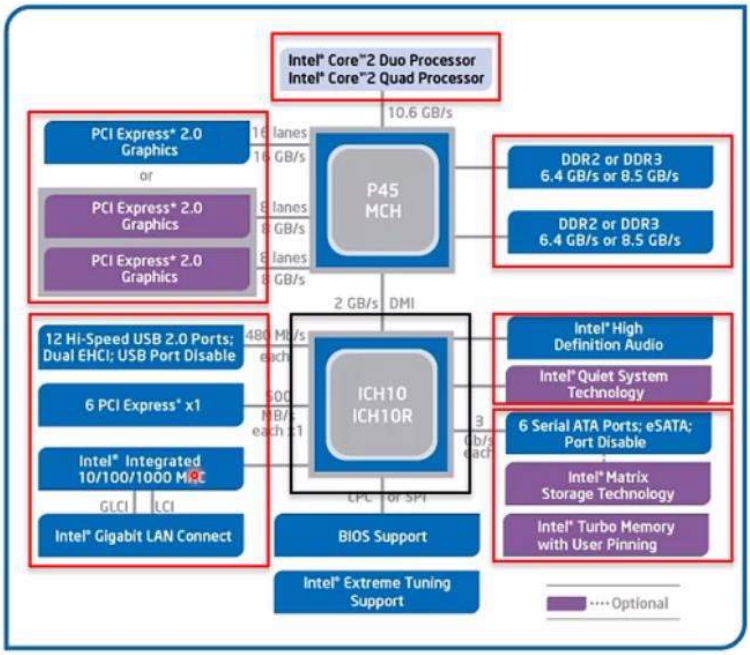
Vector Address = Vector Number × 4
= $37_{10} \times 4_{10}$
= 148_{10}

```
        ORG     $1000
T5VN    EQU     32+5            ; trap 5 vector number
T5VA    EQU     T5VN*4          ; trap 5 vector address


START   MOVE.L  #TRAP5,T5VA     ; update vector table


* Use trap 5 to clear block of 16 bytes in memory

        LEA     $9000,A1        ; start address
        LEA     $9010,A2        ; end address
        TRAP    #5              ; clear!
        NOP


*-----------------------------------------------------
* TRAP 5 Exception Handler
* Purpose: Clear block of memory
* Entry: A0 points to start of block to clear
*        A1 points to end of block to clear
* Exit:  A0 points to end of block + 1
*-----------------------------------------------------

TRAP5   CMPA.L  A2,A1
        BHS     DONE
        CLR.B   (A1)+
        BRA     TRAP5
DONE    RTE


        END     START
```

Logical View of a Traditional Computer

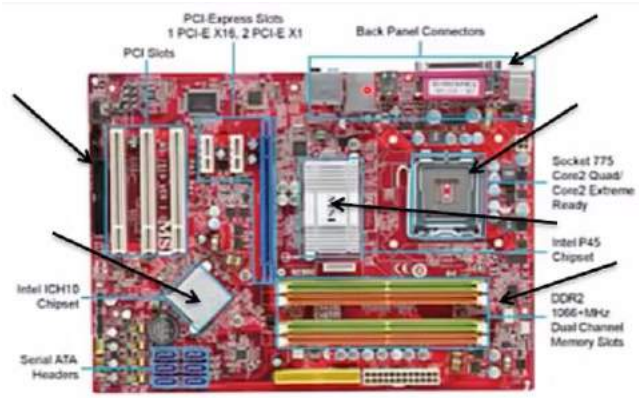


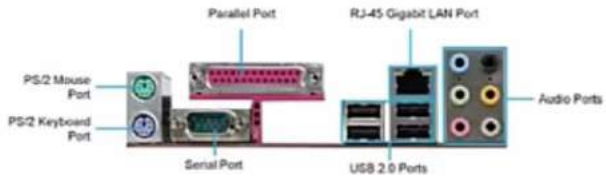
Semi-Logical View of Computer



Intel® P45 Express Chipset Block Diagram

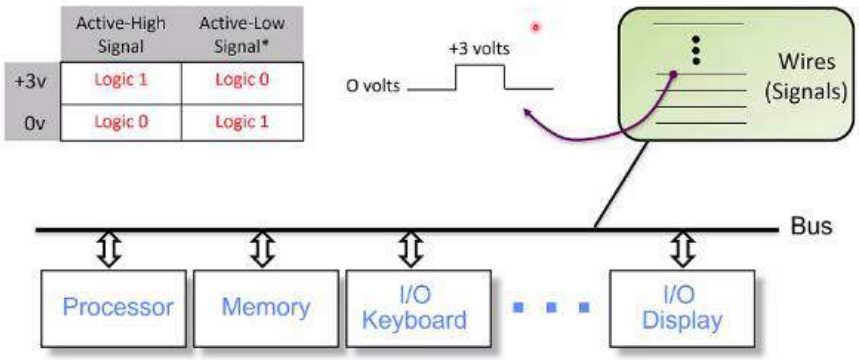
Physical View of Computer





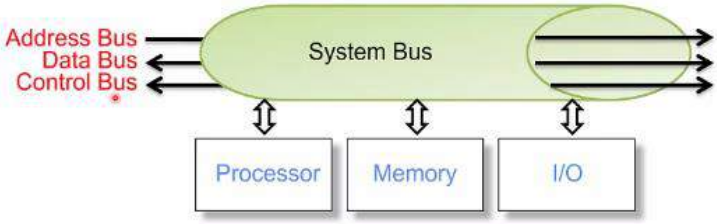
Buses

- Traditional way to connect CPU, main memory, and I/O devices
  - A set of parallel wires for carrying bits which is shared by a number of devices



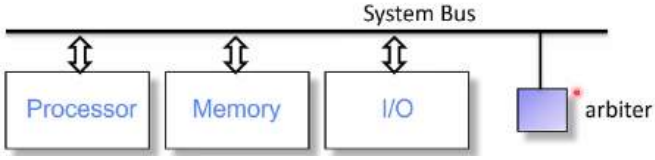
System Bus Sub-Assemblies

- The system bus consists of individual signals organized into three sub-assemblies
  - Address bus transports addresses of memory and I/O devices
  - Data bus transports data and instructions
  - Control bus transports synchronization signals



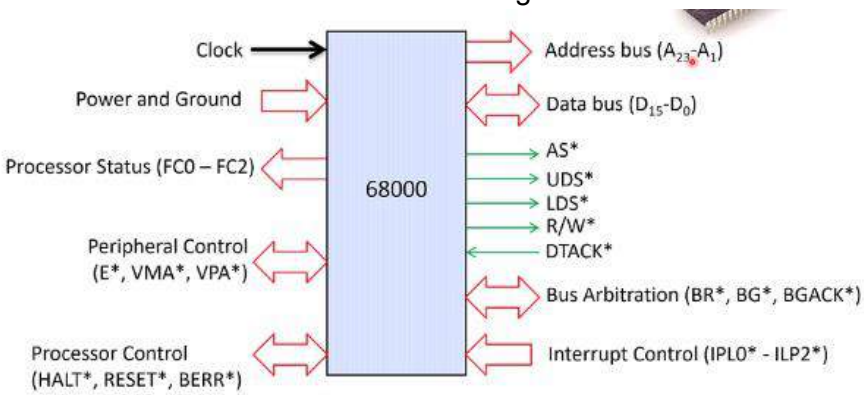
Buses: The Good, The Bad, and The Ugly

- Buses are a shared medium
  - Nice properties
    - Reduces the number of connections in the computer
    - Expansion simple
    - Broadcast, serialization
  - Drawbacks
    - Slow - one transaction at a time
    - Needs a clock that is much slower than the CPU clock
    - Electrically ugly - discontinuities limit operating speed
    - Closed-loop systems



68000 Bus Signals

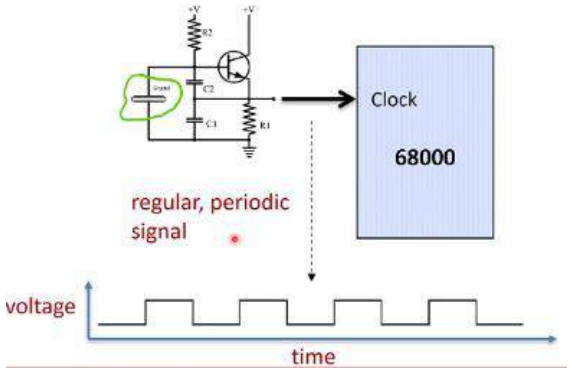
- The 68000's bus contains a total of 64 signals



Clock

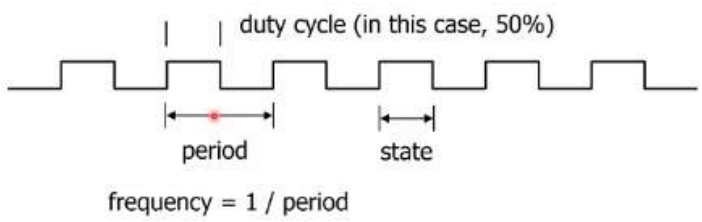
- External signal used to keep time in synchronous systems, like processors

- Forces system to make internal decision about current state or hardware configuration; sets base frequency (tempo) in computer



Clock Terminology

- Clocks are regular periodic signals
  - Clock tick - two reference edges (positive and negative)
  - Clock period - time between ticks; measured in fractions of seconds; also known as “cycle time”
  - Clock frequency - the inverse of the clock period; measured in cycles per second or Hertz (Hz)
  - Duty-cycle - time clock is high between ticks; expressed as a % of period
  - State (68000) - ½ clock cycle



Problem

- A processor is clocked at a frequency of 3 GHz
- What is the period or cycle time of the clock?

$F = 3\ GHz$
$= 3 \times 10^9 \ \frac{cycles}{second}$

$T = ?$

$T = \frac{1}{F} = \frac{cycle}{\frac{cycles}{second}} = seconds$ (units)

$T = \frac{1}{3 \times 10^9}$
$= 0.33 \times 10^{-9}$
$= 0.33$ nanoseconds

Common Units for Clock Frequency and Cycle Time

Clock Cycle Time (Clock Period) is typically measured in fractions of a second

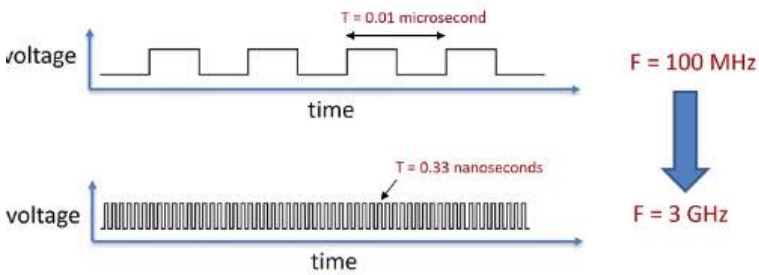
Seconds	Units
0.001s	milliseconds (ms)
0.000 001	microseconds (us)
0.000 000 001	nanoseconds (ns)

Clock frequency is measured in cycles per second (Hertz)

Cycles/Second	Units
1000	Kilohertz (KHz)
1 000 000	Megahertz (MHz)
1 000 000 000	Gigahertz (GHz)

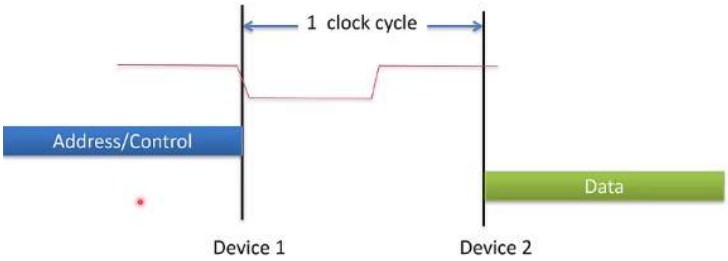
Clock Speed Can Change

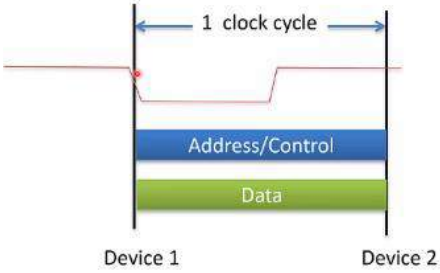
- Base clock frequency of the 68000 system described in your textbook is approximately 3.68 MHz
- Base clock frequency of a typical PC is approximately 100 MHz
    - But clock speed can be throttled up (or down) for more or less performance or power consumption



Synchronous Bus

- Includes a clock in control lines
- Fixed protocol for communication relative to the clock
- Advantages
    - Involves very little logic to implement
- Disadvantages
    - Every transfer on the bus must run at the same clock rate
    - Bus cannot be long if it is fast

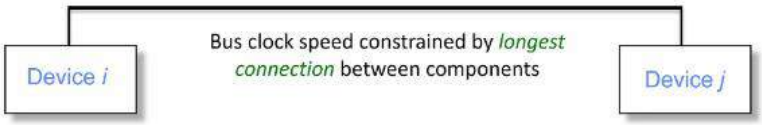




Maximum Length of Bus

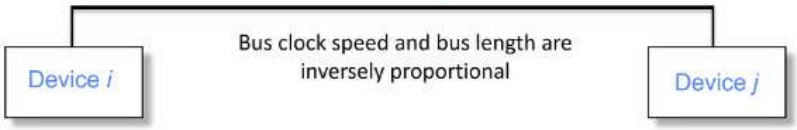
- Any signal put on the bus has to travel between any two components that share the bus in any single (Bus) clock cycle
  - Assumptions
    - Light travels at 3×10^8 m/s
    - Bus clock frequency = 3×10^9 cycles/second (Gigahertz)

$$\text{distance}_{ij} = \frac{3 \times 10^8 \text{ meters / second}}{3 \times 10^9 \text{ cycles / second}} = 0.1 \text{ meters / cycle}$$



- Assumptions
  - Same light assumption above
  - Bus clock frequency = 33×10^6 cycles/second (Megahertz)

$$\text{distance}_{ij} = \frac{3 \times 10^8 \text{ meters / second}}{33 \times 10^6 \text{ cycles / second}} = 9.1 \text{ meters / cycle}$$



Asynchronous Buses

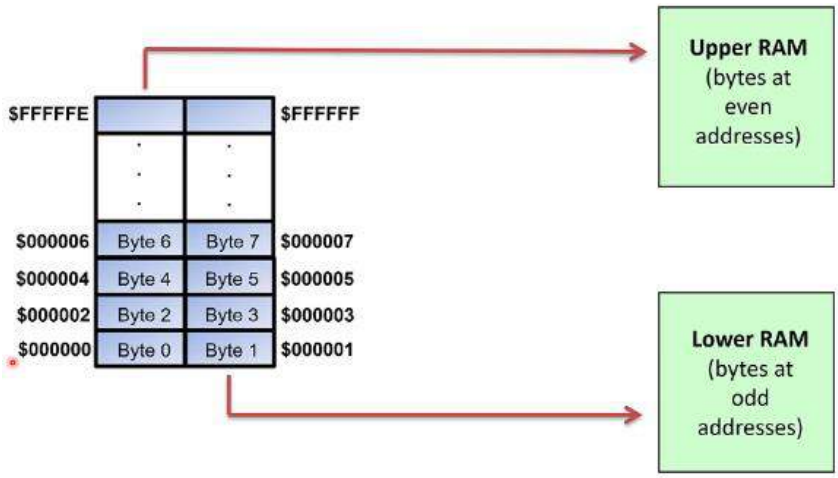
- No clock control line
  - Requires handshaking
- Advantages
  - Can easily accommodate a wide range of devices that operate at different speeds
  - No clock stew problems, so bus can be quite long
- Disadvantages
  - Potentially slower than synchronous buses, but…

Asynchronous Bus Control on the 68000

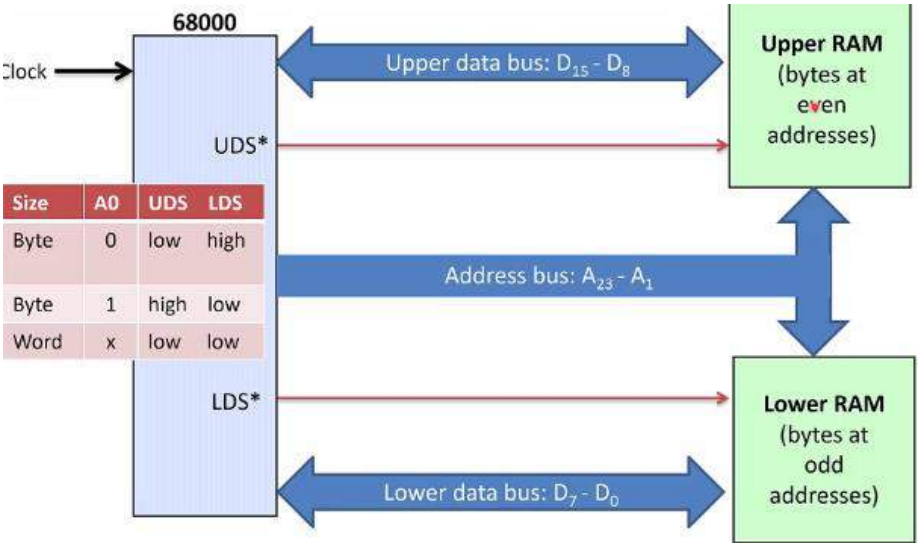
Signal	Active	Purpose	Direction
Address Strobe (AS)*	Low	Indicates a valid address exists on the address bus	Output signal
Read/Write (R/W*)	Read=high Write = low	Determines whether a read or write operation is to be performed	Output signal
Upper Data Strobe (UDS*)	Low	Controls transfer of byte (at even address) over upper byte of data bus	Output signal
Lower Data Strobe (LDS*)	Low	Controls transfer of byte (at odd address) over lower byte of data bus	Output signal
Data Acknowledgement (DTACK*)	Low	Informs processor that data transfer has completed	Input signal

* means active low

Dual Memories

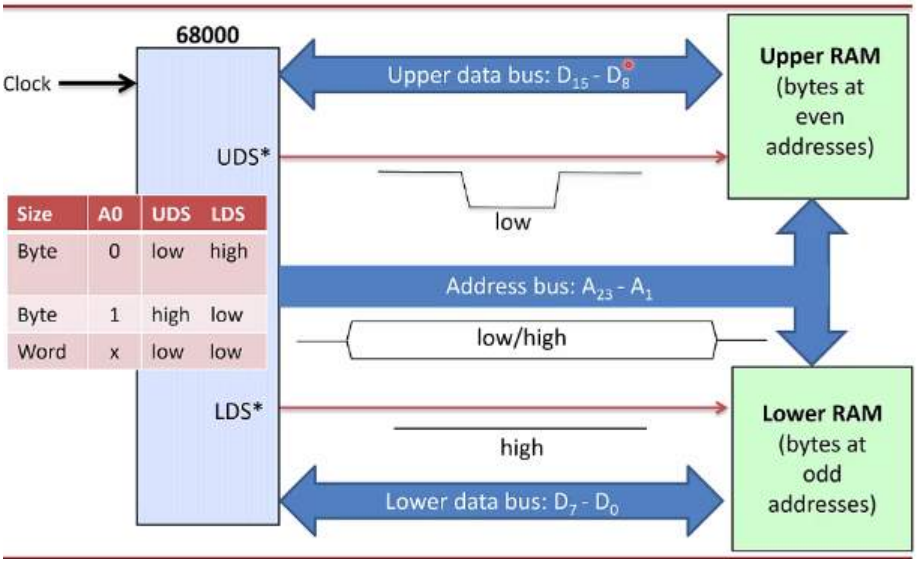


Upper and Lower RAMs/Data Buses



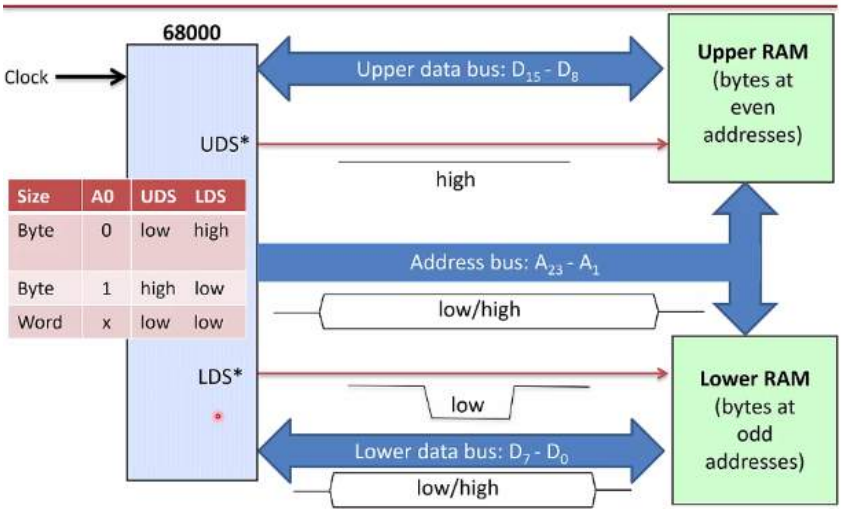
Size	A0	UDS	LDS
Byte	0	low	high
Byte	1	high	low
Word	x	low	low

MOVE.B $008000,D0



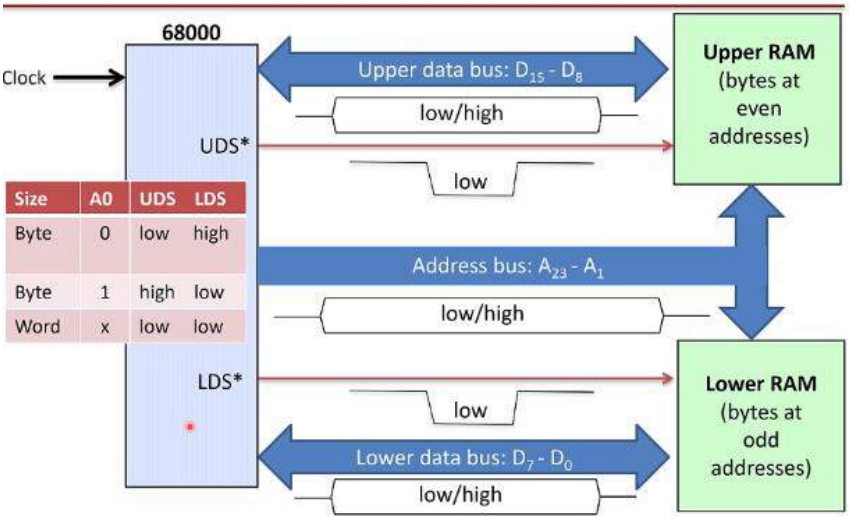
Size	A0	UDS	LDS
Byte	0	low	high
Byte	1	high	low
Word	x	low	low

MOVE.B $008001,D0



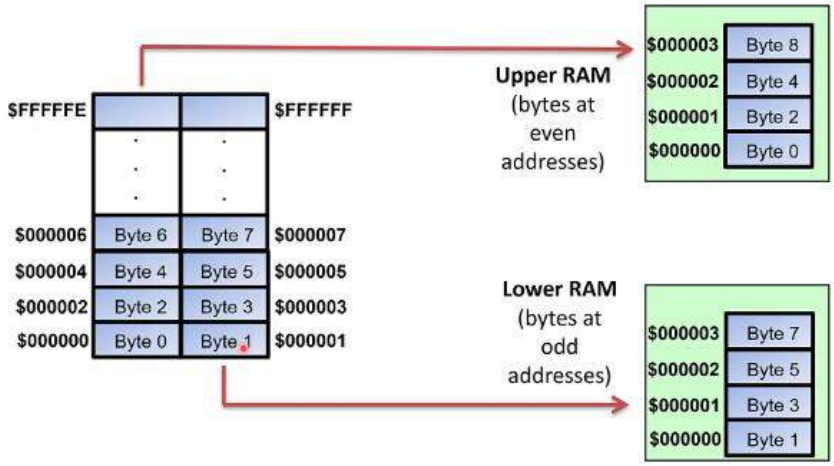
Size	A0	UDS	LDS
Byte	0	low	high
Byte	1	high	low
Word	x	low	low

MOVE.W $008000,D0



Size	A0	UDS	LDS
Byte	0	low	high
Byte	1	high	low
Word	x	low	low

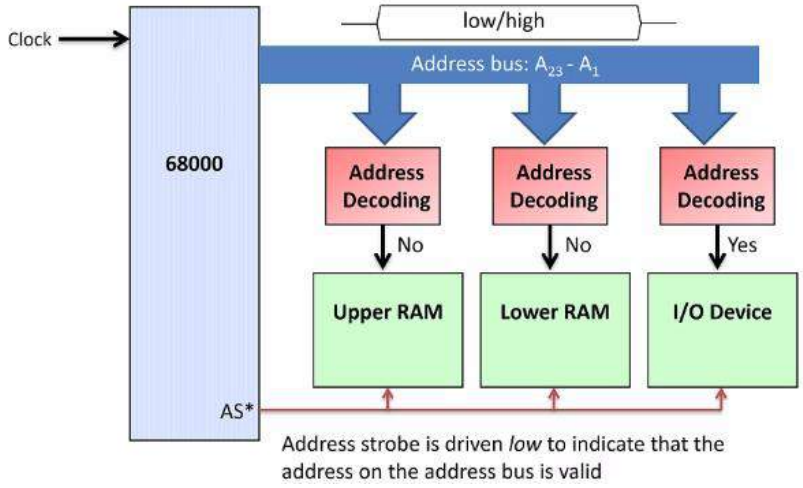
Words/Long Words Must be at Even Addresses



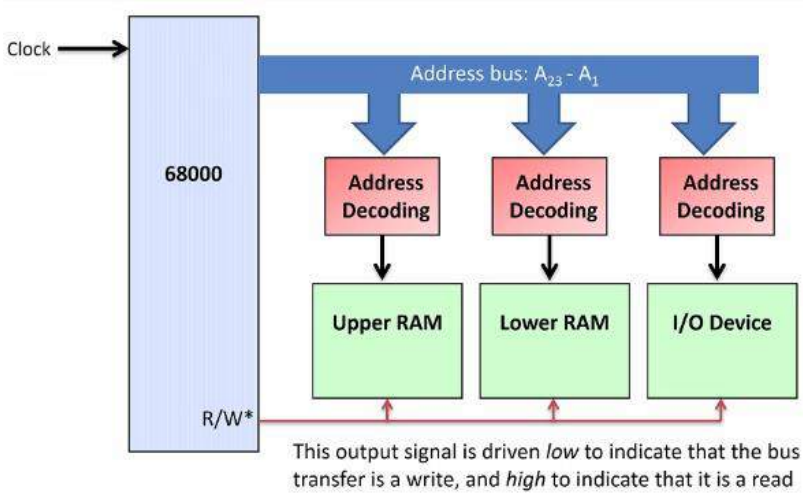
Problem

- Indicate the states (low or high) of UDS* and LDS* when the processor is involved in the following memory accesses
  - A byte written to address 3000
    - Byte
    - Even address
    - *UDS = Low
    - *LDS = High
  - A byte written to address 3001
    - Byte
    - Odd address
    - *UDS = High
    - *LDS = Low
  - A word written to address 3000
    - Word
    - Even address
    - Odd → address error exception
    - *UDS = Low
    - *LDS = Low

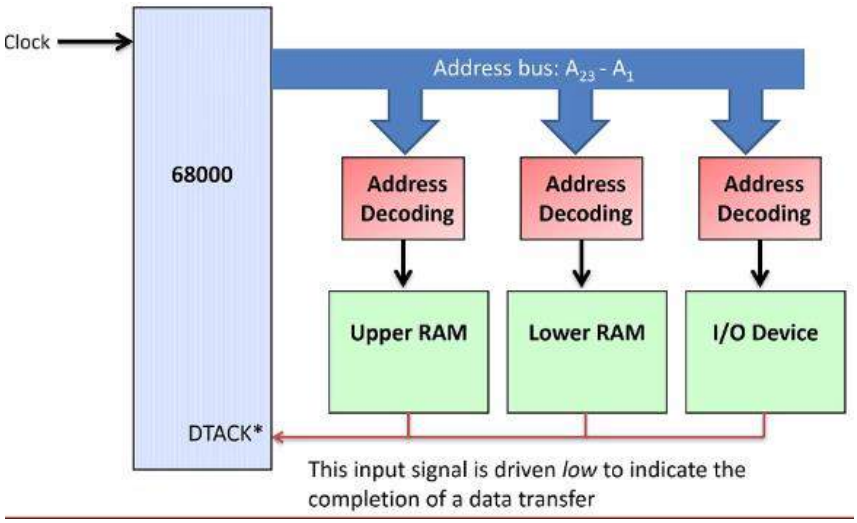
Address Strobe



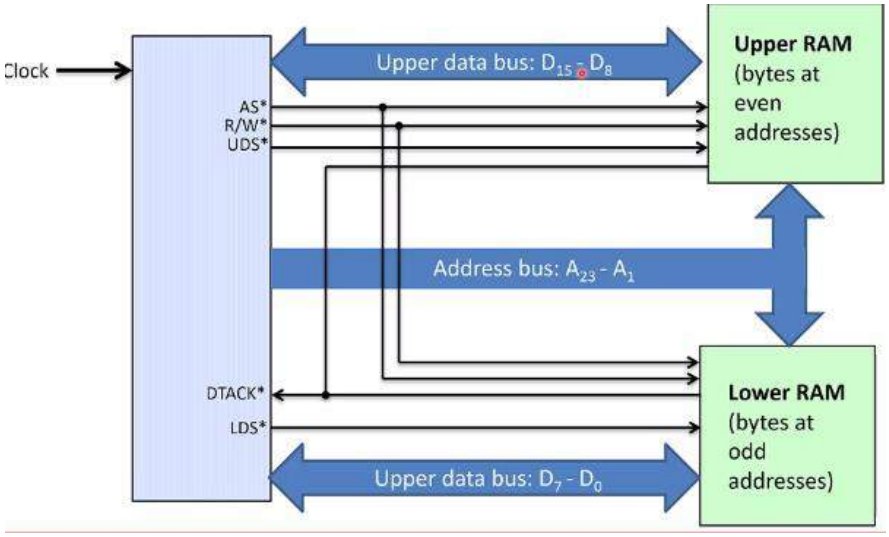
Reading and Writing



Data Transfer Acknowledge

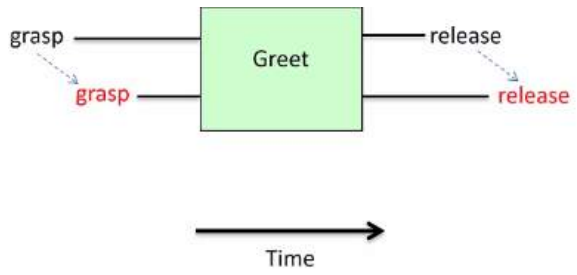


Processor/Memory Interface

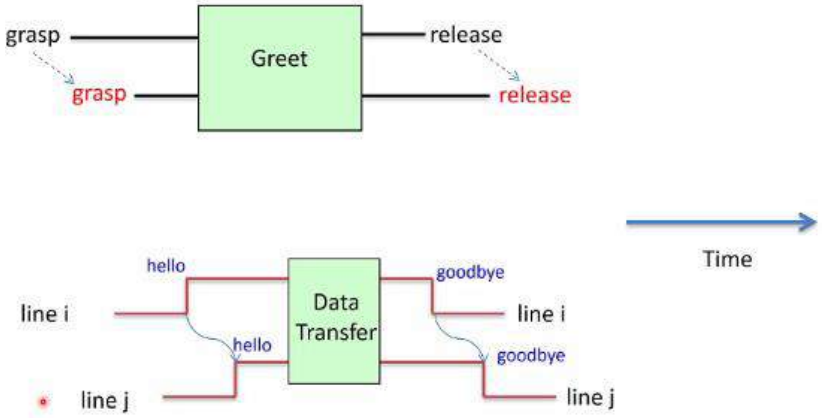


Handshaking

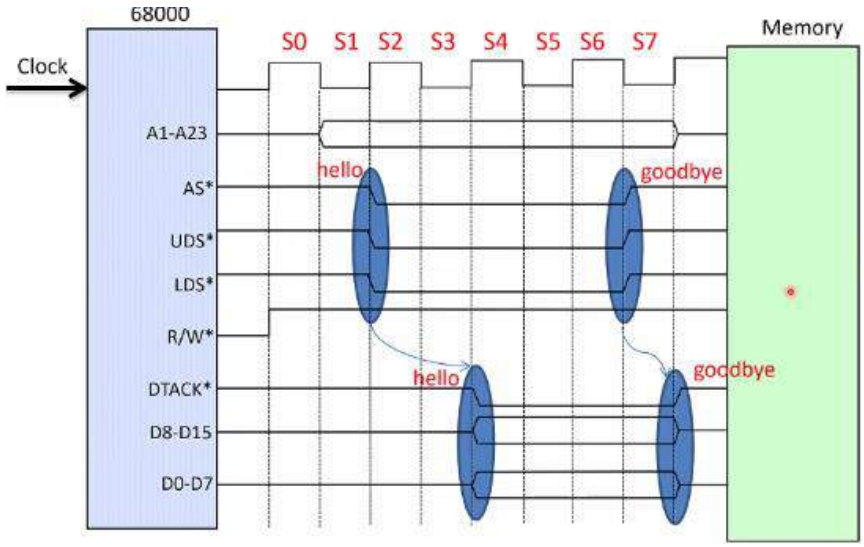
- Communication requires a protocol to structure and synchronize communication
  - Connection: start and stop communication
  - Data transfer



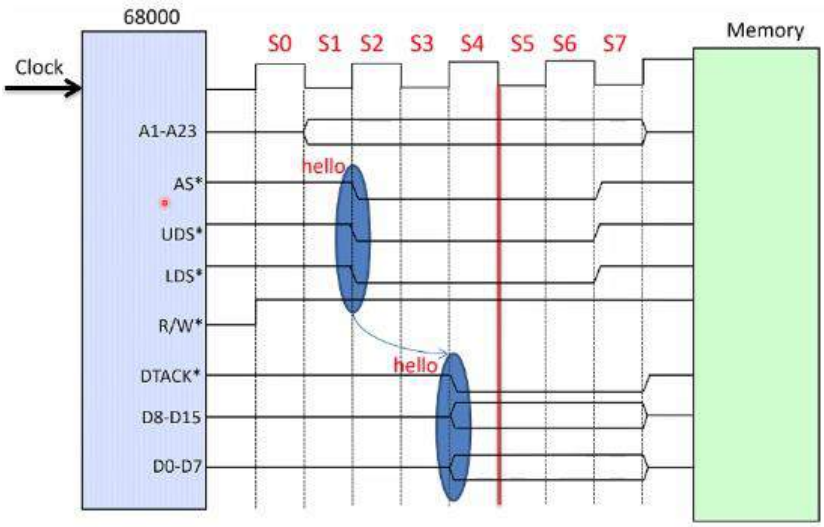
Handshaking with Hardware Devices



Read Cycle for a Word

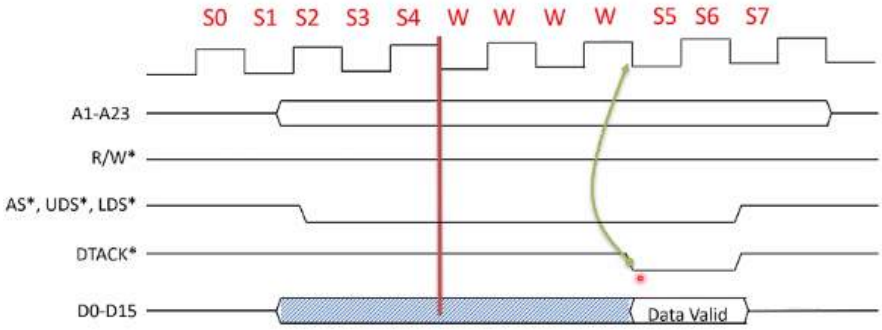


DTACK Must be Asserted Before the End of S4



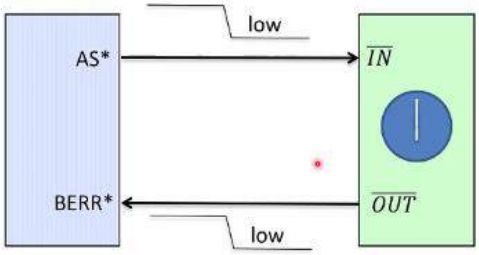
Wait States

- A wait state is an extra clocking period that lengthens the bus cycle
  - Inserted between S4 and S5 until DTACK arrives
  - Allows processor to operate with a mixture of both fast and slow components



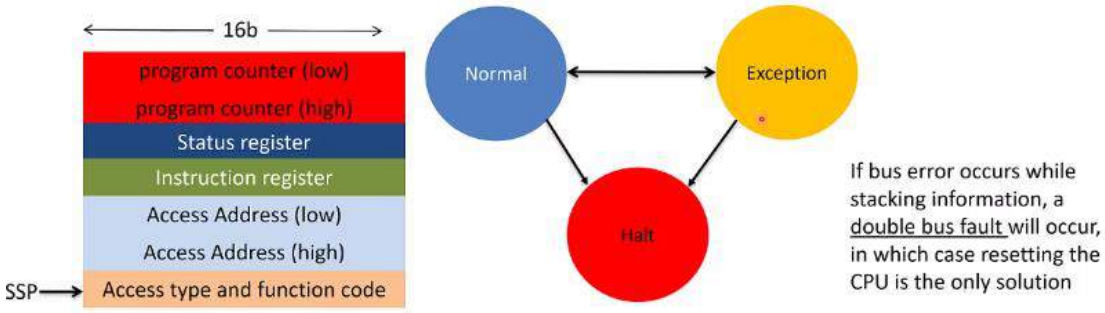
Watchdog Timer

- A watchdog timer can be used to terminate a bus cycle if DTACK never arrives



Bus Error Exception

- A bus error indicates that something has gone seriously wrong
    - More information is stored on the exception stack than is stored for either a group 1 or 2 exception



Function Codes

- Function codes indicate the type of bus cycle currently being executed

FC0	FC1	FC2	Processor Cycle Type
0	0	0	Reserved
0	0	1	User Data
0	1	0	User Program
0	1	1	Reserved
1	0	0	Reserved
1	0	1	Supervisor data
1	1	0	Supervisor program
1	1	1	Interrupt Acknowledge

Problem 1

- The CPU clock driving the bus on a particular 68000 system has a frequency of 10 Megahertz
- How long does the following instruction take to execute on the system?
- MOVE.W D0,(A0)

- operation
- data size
- address mode(s) → (r|w)

n - number of clock cycles required to execute the instruction

t_c - period of clock

MOVE.W D0, (A0)

↑ OP ↑ data size ↑ data reg direct ← indirect addressing

cycles (r/w)

n → 8(1/1)

↑ ↑ 4 clock cyles

Table D-2. Move Byte and Word Instruction Execution Times

Source	Destination								
	Dn	An	(An)	(An) +	- (An)	d(An)	d(An, ix)*	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
An	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
(An)	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
(An) +	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
- (An)	10(2/0)	10(2/0)	14(2/1)	14(2/1)	14(2/1)	18(3/1)	20(3/1)	18(3/1)	22(4/1)
d(An)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(An, ix)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
xxx.W	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
xxx.L	16(4/0)	16(4/0)	20(4/1)	20(4/1)	20(4/1)	24(5/1)	26(5/1)	24(5/1)	28(6/1)
d(PC)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(PC, ix)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
#xxx	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)

$n = 8$

$t_c = 1/f$

$= 1/10\ MHz$

$= 0.1\ \mu s$

$= 100\ ns$

Execution Time $= n \times t_c$

$= 8 \times 0.1\ \mu s$

$= 0.8\ \mu s$

Problem 2

- The CPU clock driving the bus on a particular 68000 system has a frequency of 10 Megahertz
- However, the 68000 system inserts one wait state per read or write cycle

- How long does the following instruction from problem 1 now take to execute on the system?

cycles (r/w)

10 ~~8~~ (1/1)

n

4 clock

5 cycles

Table D-2. Move Byte and Word Instruction Execution Times

Source	Destination								
	Dn	An	(An)	(An) +	- (An)	d(An)	d(An, ix)*	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
An	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
(An)	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
(An) +	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
- (An)	10(2/0)	10(2/0)	14(2/1)	14(2/1)	14(2/1)	18(3/1)	20(3/1)	18(3/1)	22(4/1)
d(An)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(An, ix)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
xxx W	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
xxx L	16(4/0)	16(4/0)	20(4/1)	20(4/1)	20(4/1)	24(5/1)	26(5/1)	24(5/1)	28(6/1)
d(PC)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(PC, ix)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
#xxx	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)

$$\text{Execution Time} = n \times t_c$$

$$= \overset{10}{\cancel{8}} \times 0.1\,\mu s$$

$$= \underset{1}{\cancel{0.8}}\ \mu s$$

Problem 3

- A certain processor and memory share a 32-bit bus running at 100 MHz
- Eight clock cycles are required to access a 32-bit value from memory
- What is the bandwidth of the bus, where bandwidth is the number of bytes that can be transferred per second over the bus

① Time to read/write a 32-bit value

$n = 8$ clock cycles

$$t_c = \frac{1}{f} = \frac{1}{100\ MHz} = 0.01 \times 10^{-6}\ (s)$$

$$= 0.01\ \mu s$$

$$= 10.0\ ns$$

$$T_{32} = n \times t_c$$

$$= 8 \times 10.0\ ns$$

$$= 80 ns$$

② Number of reads/writes per second

$$\frac{1}{T_{32}} = \frac{1}{80\ ns} = 12.5 \times 10^6\ \frac{\text{reads/writes}}{\text{second}}$$

③ Number of bytes per second

$= \#\text{read/write(s)} \times \#\text{ bytes}$

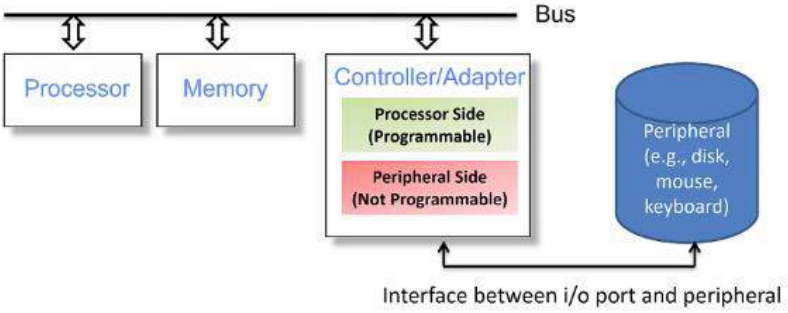
$= 12.5 \times 10^6 \ \frac{\text{reads/writes}}{\text{second}} \times \frac{32}{8} \ (\text{bytes})$

$= 50 \text{ MB/s}$

CIS2030 Microcomputers Week 10

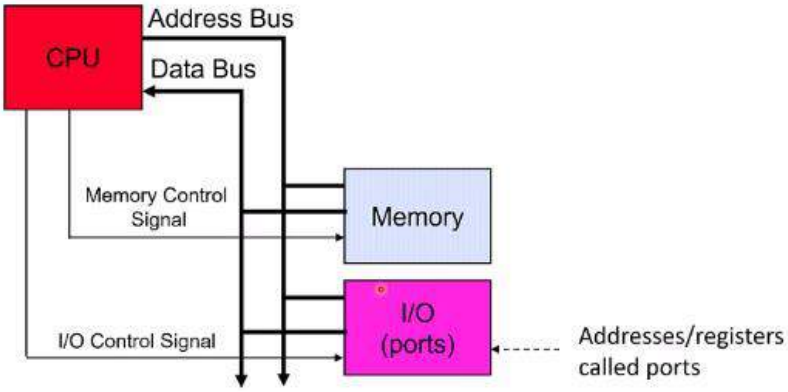
I/O Device Controllers/Adapters

- Controller/Adapter - interface between I/O device and bus
- Contains registers, and possible memory, processor, clock
- Organized into two halves
    - Processor - device driver is part of OS
    - Peripheral - external to computer



Isolated I/O

- Shared address and data bus, separate control buses for I/O and memory



Pro

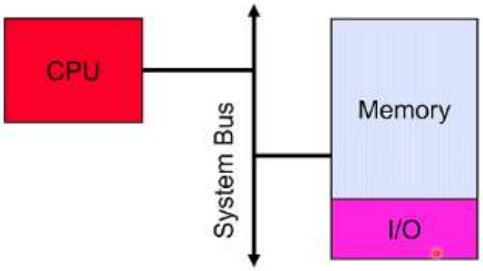
- All addresses can be used by memory

Cons

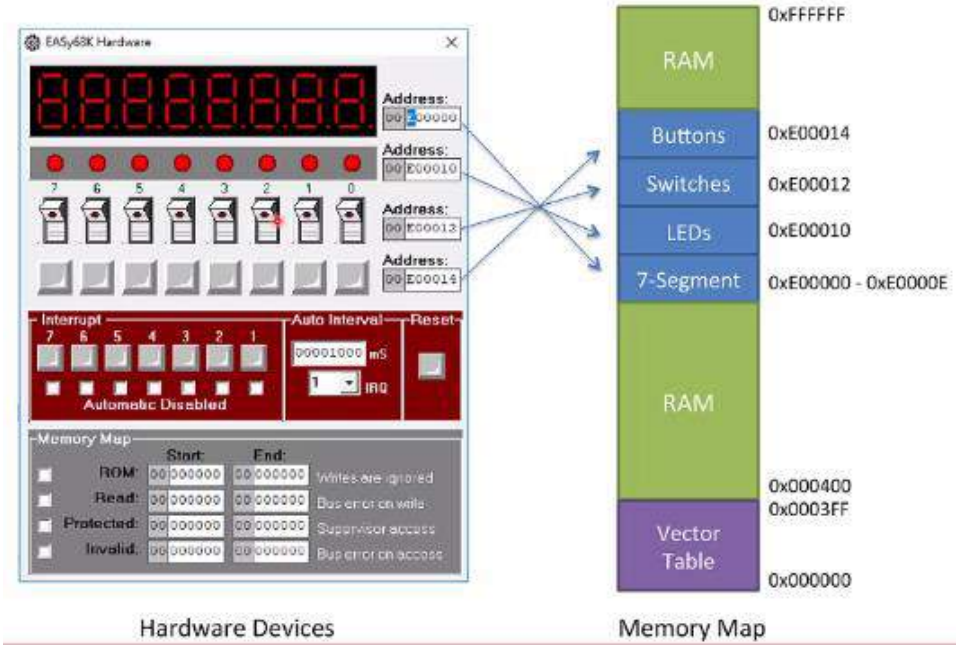
- Separate (special) instructions for accessing ports
  - `special_instruction <reg>,port`
  - `special_instruction port,<reg>`
- Complex due to separate control logic for each address space

Memory-Mapped I/O

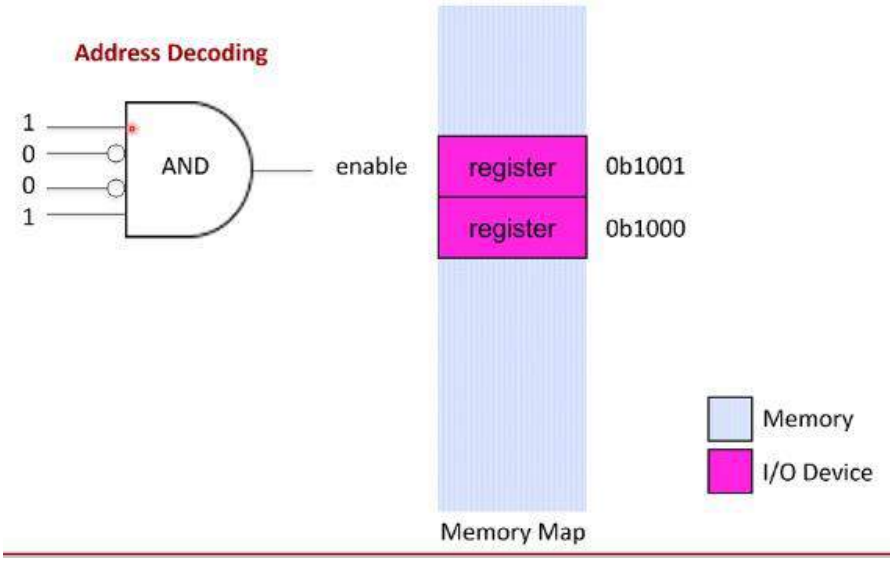
- Memory and I/O devices share the same system bus and same address space
- Data-transfer instructions can be used to move data to and from I/O device registers
  - A **load** operation moves data from an I/O device to a CPU register
  - A **store** operation moves data from a CPU register to an I/O device register



Example of Memory-Mapped I/O from Easy68K

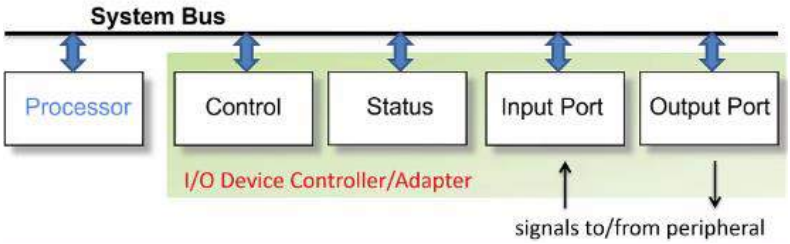


Mapping Registers into Addresses Requires Logic



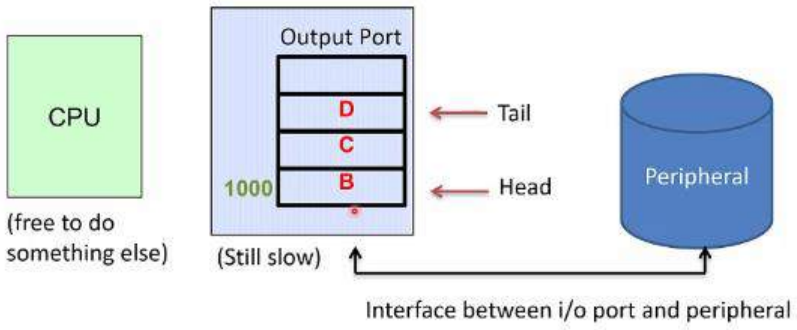
Role of Registers in I/O Controller/Adapter

- Controller Register(s)
  - Programmed (by device driver in OS) to control the operation of the peripheral
- Status Register(s)
  - Queried (by device driver in OS) to learn things about the peripheral
- Input and Output ports
  - Temporarily hold data that has been received or is about to be transmitted

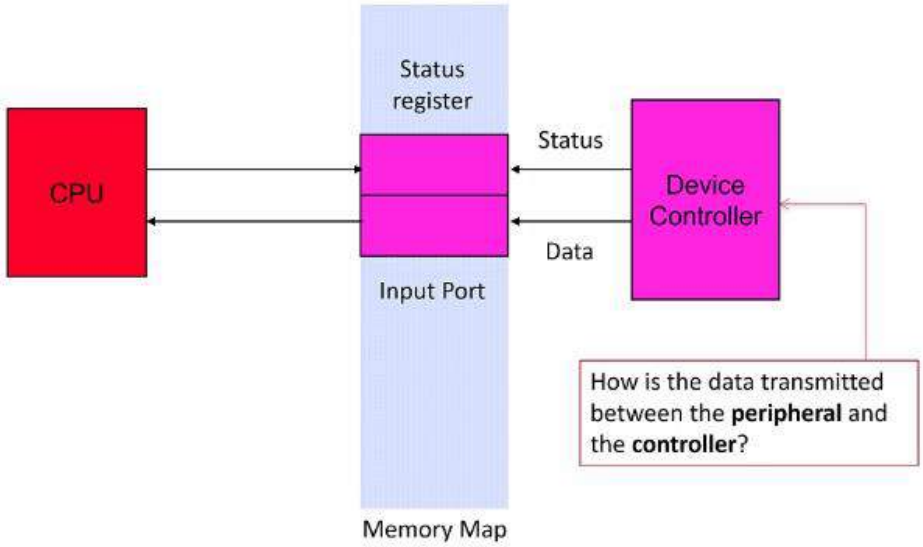


Input and Output Port Buffering

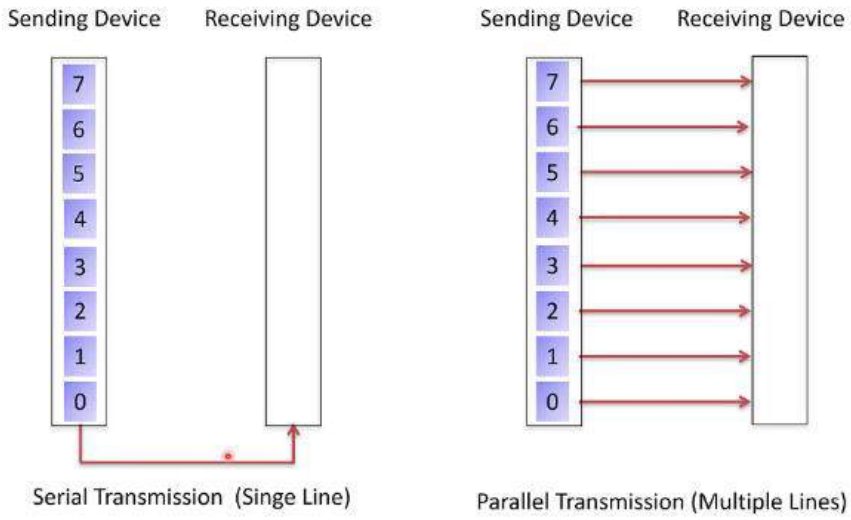
- Input and output ports are usually buffered
  - Typically implemented as a FIFO queue
  - Avoids loss of data at input port
  - Avoids processor having to wait at output port



Simple I/O Involving Single Character



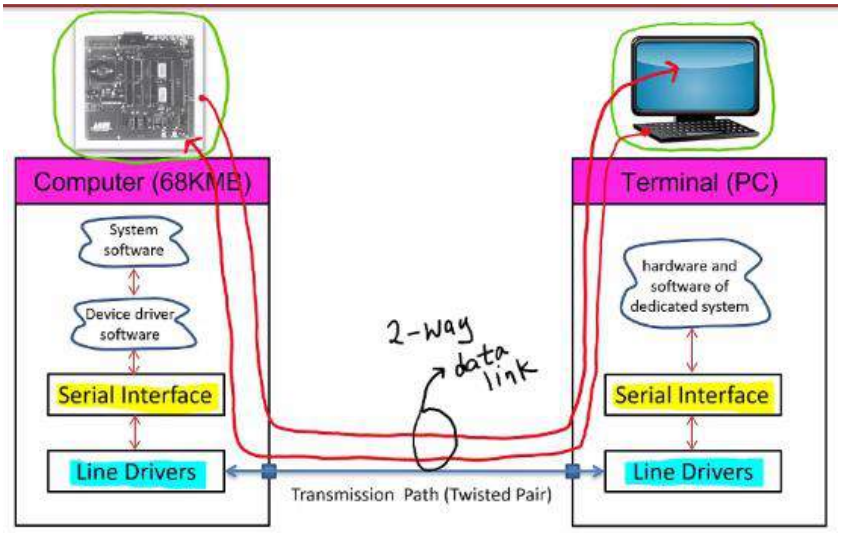
Serial/Parallel Transmission



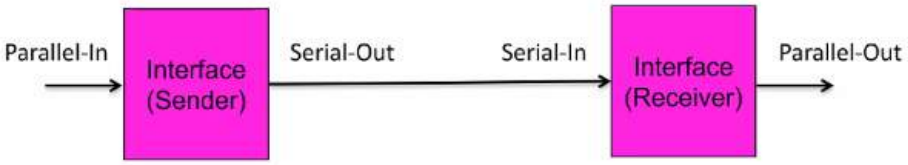
Advantages of Serial Transmission

- Advantages compared to parallel transmission
  - Cheaper
  - Less noise (e.g., cross talk)
  - Smaller interface
  - Fast (e.g., USB)

Components of Serial Data Link

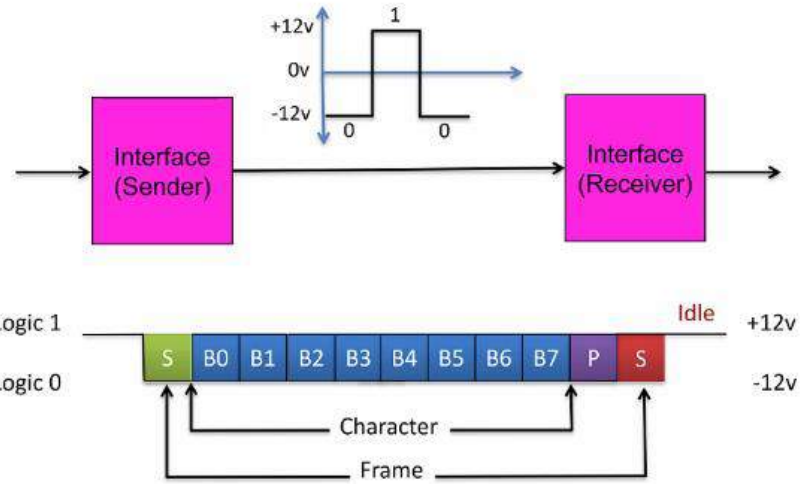


Serial Interface



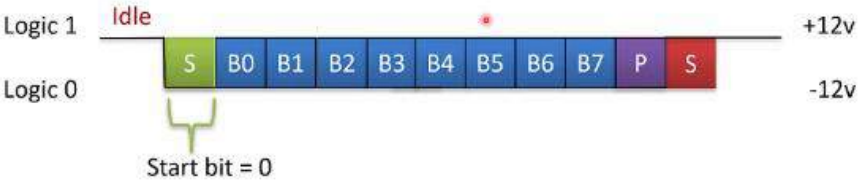
- Data must be formatted so that the receiver can recognize the beginning and end of the data
- Both the sender and receiver must agree on the formatting parameters prior to transmission

The Framing of a Character

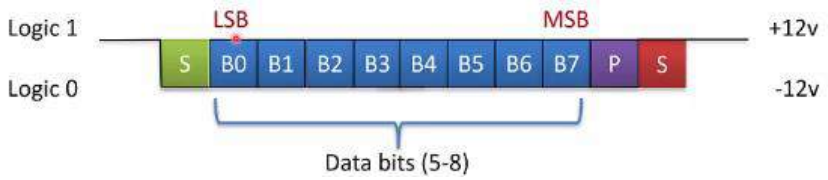


Idle Line - No Transmission

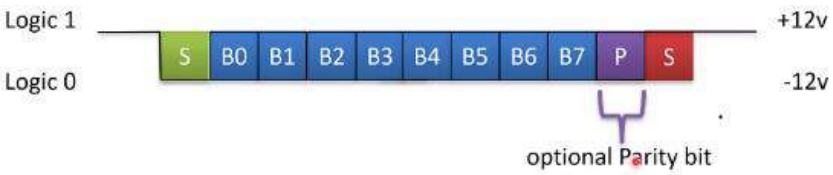
Start Bit



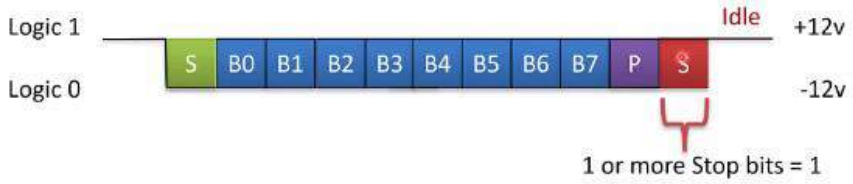
Data Bits (Character)



Parity Bit (Control)



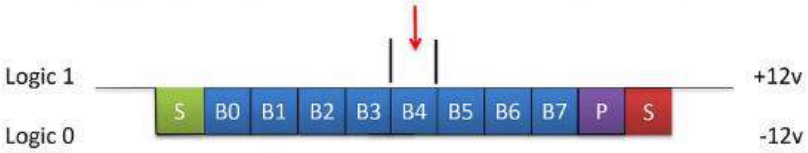
Stop Bits (Control)



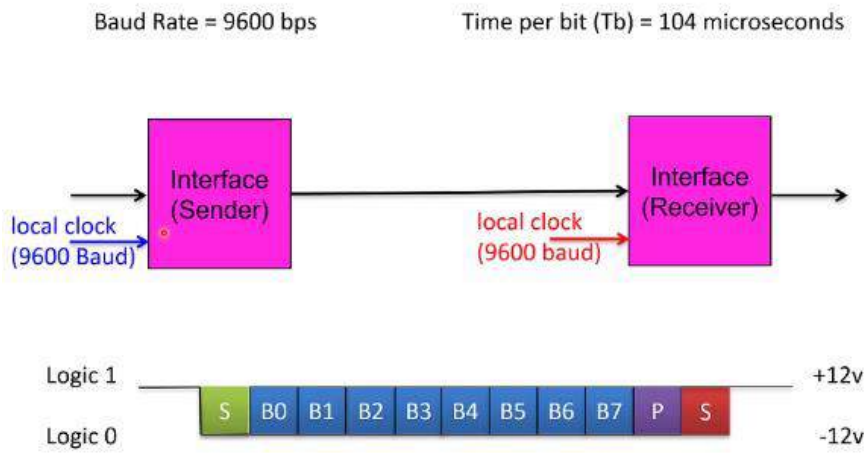
Baud Rate

- The speed (or frequency) with which bits are transmitted is defined as the **baud rate** and is measured in bits per second

Baud Rates (bps)	Time-Per-Bit (Tb = 1 / Baud Rate)
1200	833 microseconds
2400	416 microseconds
4800	208 microseconds
9600	104 microseconds
115,200 (max)	8.6 microseconds



Local Clocks



Problem

- An asynchronous serial interface is configured to operate as follows
  - 9600 baud
  - 1 start bit
  - 7 data bits
  - 1 parity bit
  - 2 stop bits
- How much time would it take to transmit 1000 ASCII characters?

$T_b = \frac{1}{9600}$ seconds

$S_c = (1+7+1+2)$
$= 11$ bits

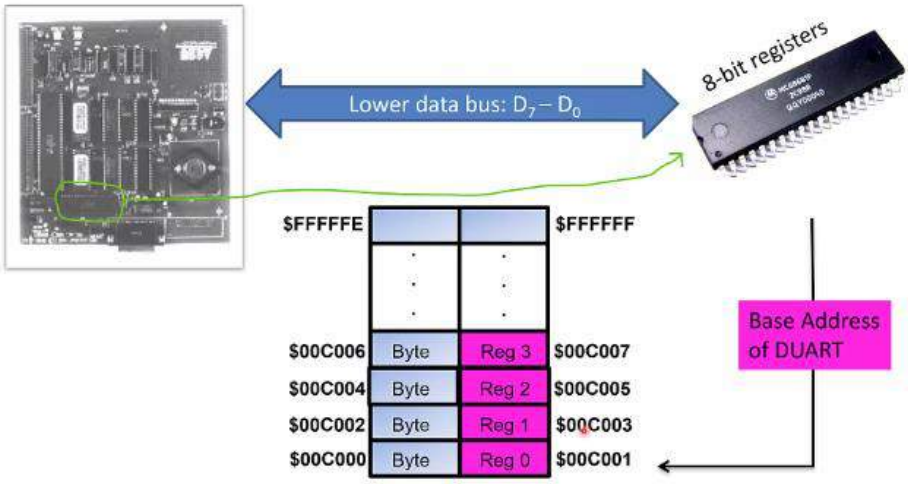
$N_c = 1000$

Transmission Time $= N_c \times S_c \times T_b$

$1000 \times 11 \times \frac{1}{9600}$

1.14 seconds

DUART - Memory Mapped Base Address



DUART's Internal Registers

	Read Operation		Write Operation	
Address	**Mnemonic**	**Full Name**	**Mnemonic**	**Full Name**
C001	MR1A, MR2A	Mode Register A	MR1A, MR2A	Mode Register A
C003	SRA	Status Register A	CSRA	Clock-Select Register A
C005	N/A	Do not access	CRA	Command Register A
C007	RBA	Receive Buffer A	TBA	Transmit Buffer A
C009	IPCR	Input Port Change Register	ACR	Auxiliary Control Register
C00B	ISR	Interrupt Status Register	IMR	Interrupt Mask Register
C00D	CUR	Counter Upper Register	CTUR	Counter/Timer Upper Register
C00F	CLR	Counter Lower Register	CTLR	Counter/Timer Lower Register

	Read Operation		Write Operation	
Address	**Mnemonic**	**Full Name**	**Mnemonic**	**Full Name**
C011	MR1B, MR2B	Mode Register B	MR1B, MR2B	Mode Register B
C013	SRB	Status Register B	CSRB	Clock-Select Register B
C015	N/A	Do not access	CRB	Command Register B
C017	RBB	Receive Buffer B	TBB	Transmit Buffer B
C019	IVR	Interrupt Vector Register	IVR	Interrupt Vector Register
C01B	IP	Input port	OPCR	Output Port Configuration Register
C01D	START	Start Counter Command	OPRSET	Output Port Register Bit Set
C01F	STOP	Stop Counter Command	CTLR	Output Port Register Bit Clear

Channel A Mode Register 1 (MR1A)

- Defines some of the basic operating parameters of the DUART

Other Parameters			Parity Mode		Parity Type	Bits/Character	
7	6	5	4	3	2	1	0
0	0	0	00 = with parity		0 = even	10 = 7	
Assume			01 = no parity		1 = odd	11 = 8	

- Configure the serial port on the DUART to operate with 7 data bits and odd parity
  - Memory-mapped address: $C001
  - MOVE.B #6,$C001

Channel A Mode Register 2 (MR2A)

- Also defines some of the basic operating parameters of the DUART

Channel Mode		Other parameters		Stop-bit Length			
7	6	5	4	3	2	1	0
0	0	0	0	0 1 1 1 = 1 bit			
Normal		Assume		1 1 1 1 = 2 bits			

- Configure the serial port on the DUART to operate normally with 2 stop bits
  - Memory-mapped address: $C001
  - MOVE.B #$F,$C001
  - MR1A and MR2A share the same address

Clock-Select Register for Channel A (CSRA)

- The clock-select register is used to specify the transmitter and receiver baud rates

Receiver Clock				Transmitter Clock			
7	6	5	4	3	2	1	0
0100	300			0100	300		
0101	600			0101	600		
0110	1200			0110	1200		
1000	2400			1000	2400		
1001	4800			1001	4800		
1011	9600			1011	9600		
1100	19200			1100	19200		
Baud Rate				Baud Rate			

- Configure the serial port on the DUART to operate at 9600 baud
  - Memory-mapped address: $C003
  - MOVE.B #$BB,$C003

Channel A Command Register (CRA)

- The clock-select register is used to control the DUART's transmitter and receiver

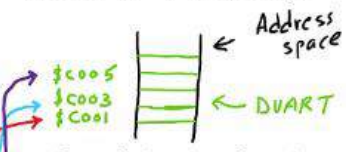
Other Commands				Transmitter Commands		Receiver Commands	
7	6	5	4	2	3	1	0
0	0	0	0	00 = no action 01 = enable 10 = disable		00 = no action 01 = enable 10 = disable	
Assume							

- Enable the DUART's transmitter and receiver
  - Memory-mapped address: $C005
  - MOVE.B #5,$C005

Device Driver for Serial Interface

- Write a device driver to initialize the DUART's channel A serial interface to operate with 1 start bit, 7 data bits, odd parity, and 2 stop bits

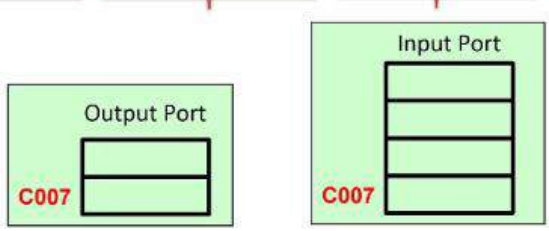
```
duart   equ     $C001                   ;base (1st) register
mr1a    equ     0
mr2a    equ     0
csra    equ     2
cra     equ     4

        lea     duart,a0                ;a0 points to duart
        move.b  #$06,mr1a(a0)           ;7 data bits, odd parity
        move.b  #$0f,mr2a(a0)           ;2 stop bits
        move.b  #$bb,csra(a0)           ;baud rate = 9600
        move.b  #$05,cra(a0)            ;enable tx/rx
```



Channel A Status Register

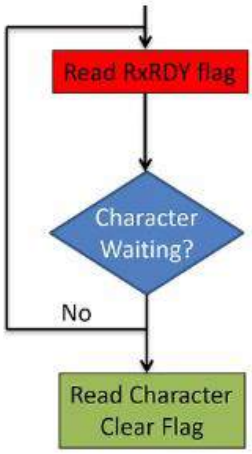
- The status register provides information about the DUART's receiver and transmitter

Received Break	Framing Error	Parity Error	Overrun Error	TxEMPTY	TxRDY	FFULL	RxRDY
7	6	5	4	3	2	1	0
0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes



- Memory-mapped address: $C003

Using a Polling Strategy to Read a Character

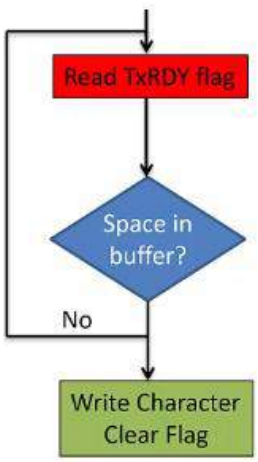


Received Break	Frame Error	Parity Error	Overrun Error	TxEMPTY	TxRDY	FFULL	RxRDY
7	6	5	4	3	2	1	0
0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes

```
duart   equu    $C001
sra     equ     2
rba     equ     6

        lea     duart,a0
loop    move.b  sra(a0),d7
        andi.b  #1,d7
        beq     loop
        move.b  rba(a0),d0
```

(SRA) B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
(mask) 0 0 0 0 0 0 0 1
AND
0 0 0 0 0 0 0 B_0

Using a Polling Strategy to Write a Character



Received Break	Frame Error	Parity Error	Overrun Error	TxEMPTY	TxRDY	FFULL	RxRDY
7	6	5	4	3	2	1	0
0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes	0 = no 1 = yes

```
duart   equ     $C001
sra     equ     2
tba     equ     6

        lea     duart,a0
loop    move.b  sra(a0),d7
        andi.b  #4,d7
        beq     loop
        move.b  d0,tba(a0)
```

(SRA) B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
(mask) 0 0 0 0 0 1 0 0
AND
0 0 0 0 0 B_2 0 0

Problem

- Write an application program that uses the device driver for the serial interface to display the message “I/O is fun!” on the terminal attached to the DUART’s serial port A
    - Assume that the serial interface is both configured and enabled, and that the previous polling routine is available for transmission
    - Display the message 10 times on 10 separate lines

Main Function

```
duart  equ    $c001                ;DUART base address
sra    equ    2                    ;status register A
tba    equ    6                    ;output port A
cr     equ    $d                   ;ASCII carriage return
lf     equ    $a                   ;ASCII line feed

** Main function **

       org    $8000
main   move.b #10,d0               ;loop counter
       lea    msg,a0               ;pointer to message
       jsr    display              ;send message
       subq.b #1,d0                ;decrement counter
       bne    main                 ;finished?
       bra    *
msg    dc.b   'I/O is fun!'cr,lf,0
```

a0

Display Function

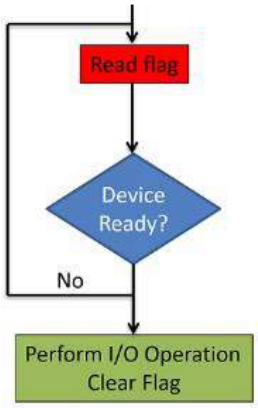
```
** display null-terminated string on terminal
** entry: a0 points to string
** exit:  a0 points to null-terminator + 1
** uses:  send function in serial interface driver

display:
         move.l     d0,-(sp)          ;save d0
loop     move.b     (a0)+,d0          ;get character
         beq  exit                    ;if null, done
         jsr  send                    ;send character
         bra  loop                    ;do it again
exit     move.l    (sp)+,d0           ;restore d0
         rts
```

Send Function (Device Driver)

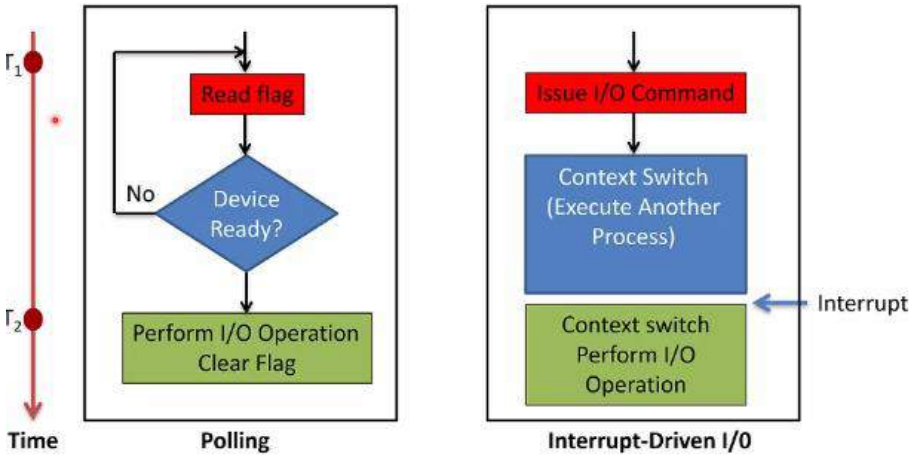
```
** send character to serial channel A
** entry: d0 (byte) contains character
** exit:  no changes to any registers

send:
         movem.l    a0/d7,-(sp)       ;save a0 and d7
         lea        a0,duart          ;point to DUART
poll     move.b     sra(a0),d7        ;read port status
         andi.b     #4,d7             ;buffer empty?
         beq        poll              ;no - try again
         move.b     d0,tba(a0)        ;send to port
         movem.l   (sp)+,a0/d7        ;restore d7 and a0
         rts
```

Pros and Cons of Polling

- Pros
  - Simple to implement in both software and hardware
- Cons
  - Occupies processor, preventing “useful” work
  - Very inefficient for data transfer (sending data to memory or sending data from memory via processor)
  - May miss important events when busy polling inactive devices or when doing something useful (and not polling)
- Summary
  - Micromanagement is too inefficient if the system is busy or has multiple devices



Interrupt-Driven I/O



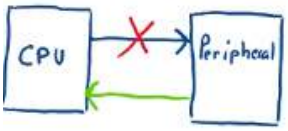
Pros and Cons of Interrupt-Driven I/O

- Pros
  - CPU can perform useful work (e.g., think of telephone)

    - CPU can’t miss important events
- Cons
    - Very inefficient for data transfer (time to process exception and perform I/O)

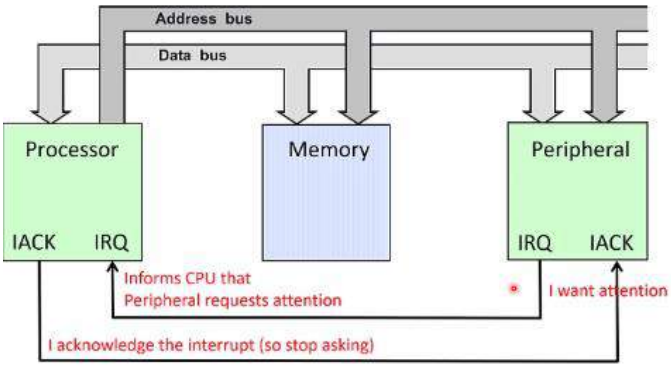
Interrupts in General

- An interrupt is an external, asynchronous signal to alert the operating system that some event or device needs attention
    - Causes may include
        - I/O completions
        - External events
            - Like mouse movement
            - Network interface card (NIC)
            - Keyboard
            - Sensors, switches
            - Etc
        - Timer expires on motherboard
            - Multi-tasking
        - Page fault
            - Virtual memory
        - Power failure
            - Shutdown system in orderly fashion
        - Many more…



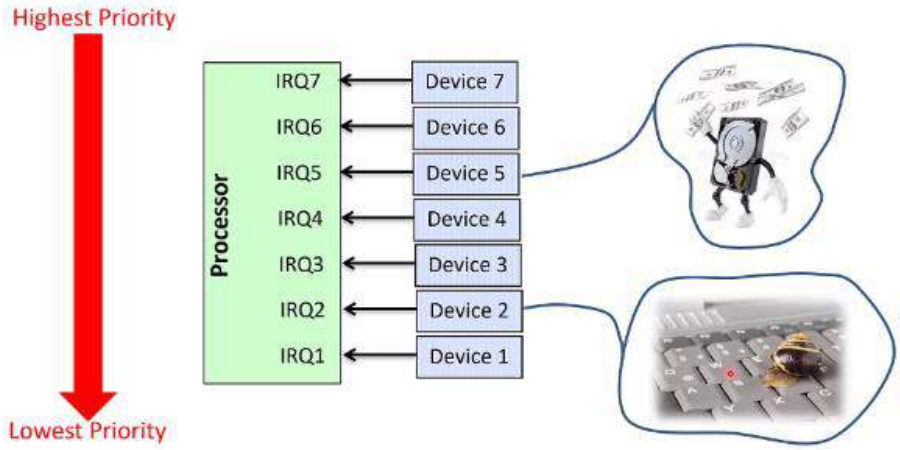
Interrupt Request Line

- Even the simplest processors have an Interrupt-Request (IRQ) line (wire) that is used by the peripherals to request attention



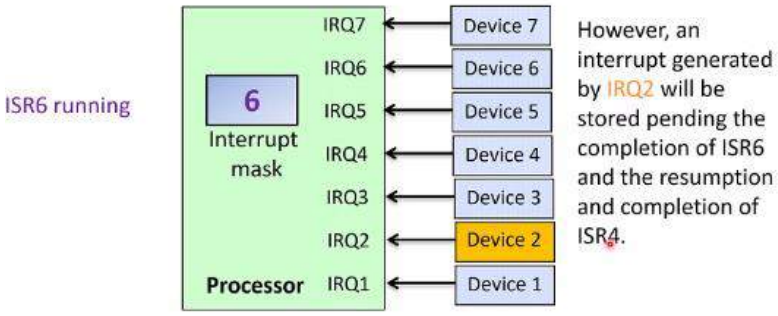
Multiple (Prioritized) Interrupts

- Most computers have more than one interrupt request input with different priorities; the processor responds to the highest priority



When are Interrupts Acknowledged?

- The interrupt mask sets the level at or below which the interrupt will not be acknowledged
    - If a device interrupts at level p, and p is greater than the current level in the mask, the mask gets set to level p

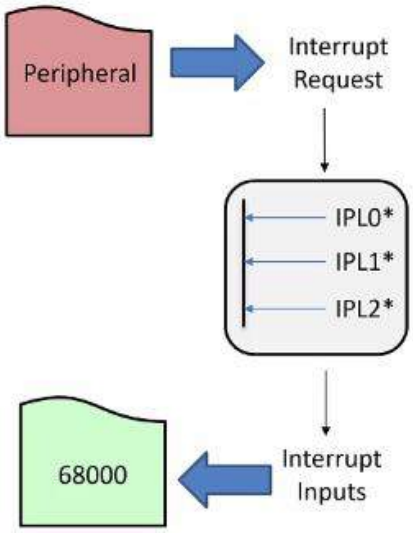


Summary

- Interrupts are prioritized
    - User programs can always be interrupted since they run with the interrupt mask set to zero
    - A lower priority interrupt-service routine can always be interrupted by a higher-priority interrupt request
    - A higher priority interrupt-service routine cannot be interrupted by a lower-priority interrupt request

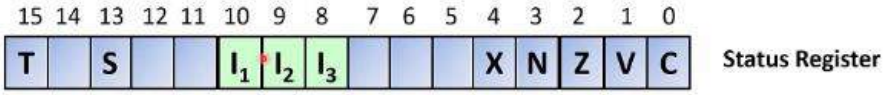
- The interrupt mask sets the level at or below which the interrupt will not be acknowledged

Interrupt Request Signals



Priority Level	IPL2*	IPL1*	IPL0*	Comment
7	Lo	Lo	Lo	Highest
6	Lo	Lo	Hi	
5	Lo	Hi	Lo	
4	Lo	Hi	Hi	
3	Hi	Lo	Lo	
2	Hi	Lo	Hi	
1	Hi	Hi	Lo	Lowest
0	Hi	Hi	Hi	None

Interrupt Mask

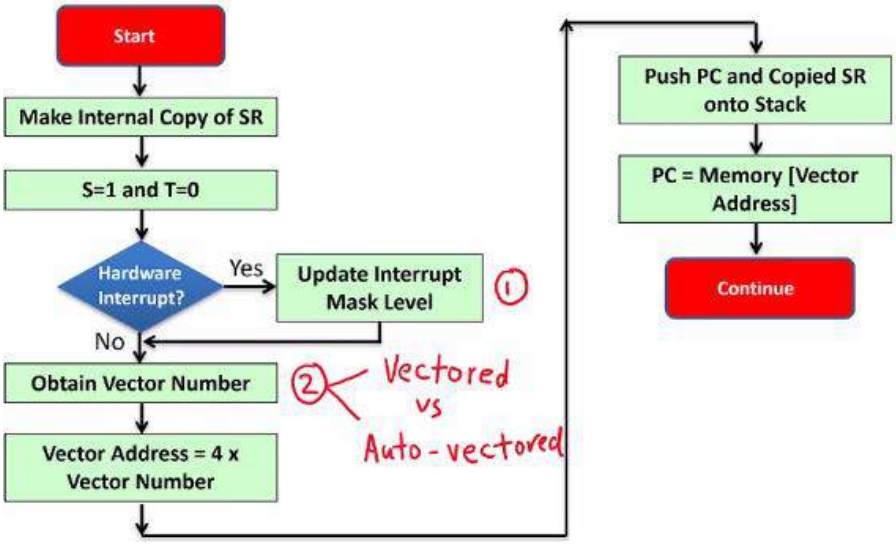


IRQ	Interrupt Mask	Recognized Levels
7 (highest)	111	7
6	110	7
5	101	6,7
4	100	5,6,7
3	011	4,5,6,7
2	010	3,4,5,6,7
1 (lowest)	001	2,3,4,5,6,7
No interrupt	000	1,2,3,4,5,6,7

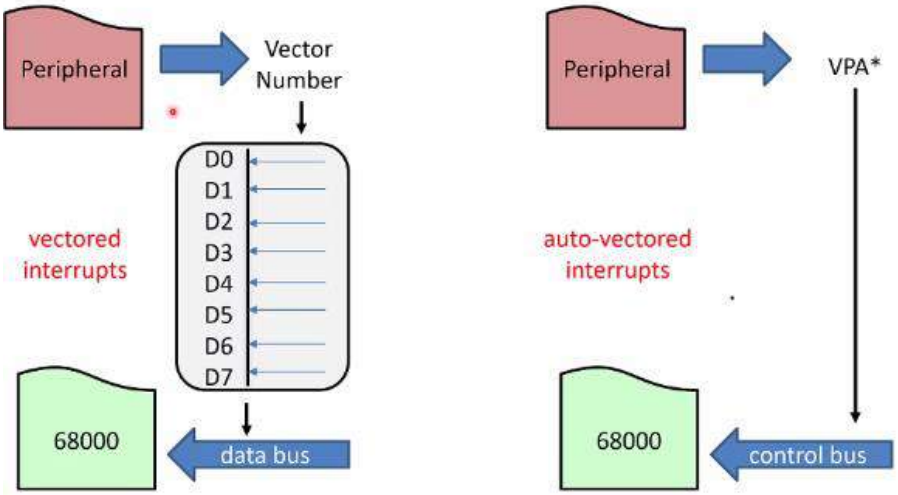
Non-maskable interrupt

All interrupts are recognized. User programs run with mask 000

How are Interrupts and Exceptions Processed?

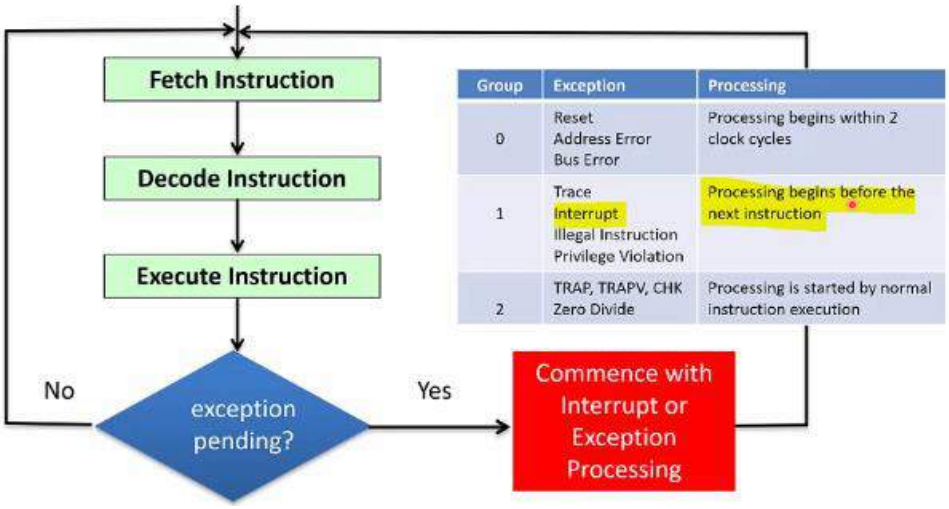


Vectored Vs. Auto-Vectored Interrupts



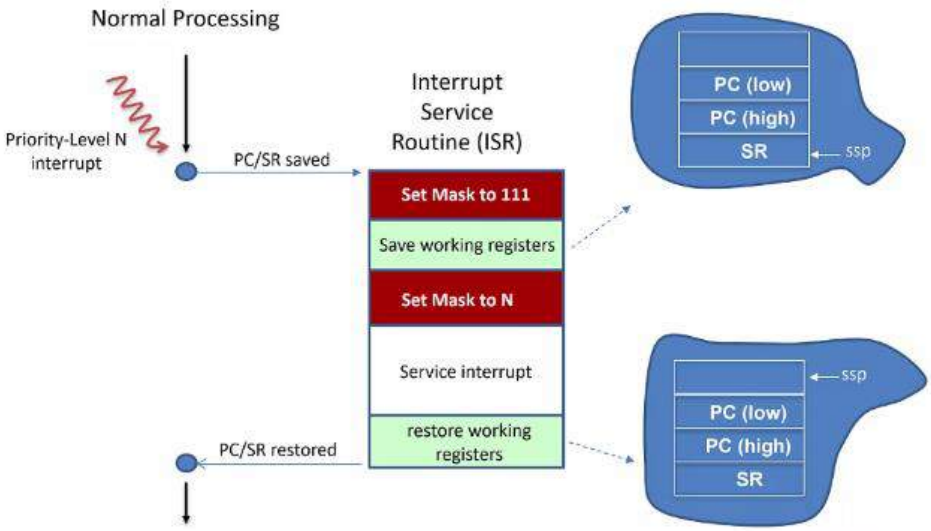
25	0064	Level 1 interrupt autovector
26	0068	Level 2 interrupt autovector
27	006C	Level 3 interrupt autovector
28	0070	Level 4 interrupt autovector
29	0074	Level 5 interrupt autovector
30	0078	Level 6 interrupt autovector
31	007C	Level 7 interrupt autovector
64-255	0100-03FF	user interrupt vectors

When are Interrupts Processed?



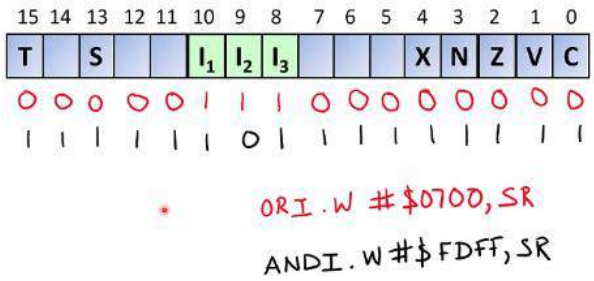
Group	Exception	Processing
0	Reset Address Error Bus Error	Processing begins within 2 clock cycles
1	Trace Interrupt Illegal Instruction Privilege Violation	Processing begins before the next instruction
2	TRAP, TRAPV, CHK Zero Divide	Processing is started by normal instruction execution

How are Interrupts Serviced?

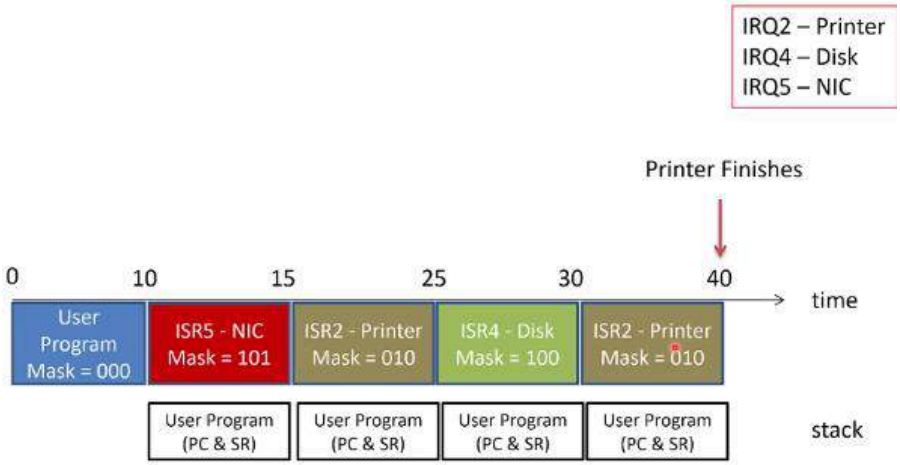


Problem

- Show how you would set the interrupt mask to level 5 without changing any of the other bits in the status register
- Assume that the processor is in the supervisor state

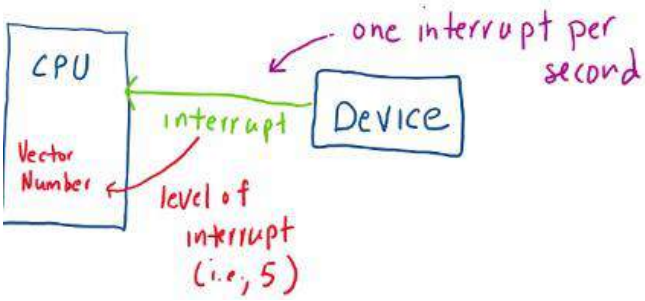


Time Sequence Showing Multiple Interrupts



Problem

- Write a level 5 autovector interrupt-service routine to display the message “Interrupts are cool!”
- Then use a timer to cause a level 5 interrupt to occur once every second



```
START:

** UPDATE VECTOR TABLE

        move.l  #isr5,$74       ; address of isr5

** DO SOMETHING WHILE INTERRUPTS ARE BEING GENERATED

here    bra     here

** LEVEL-5 AUTOVECTOR

isr5    ori.w       #$0700,sr       ; mask = 111
        movem.l     a1/d0,-(sp)     ; save working registers
        andi.w      #$fdff,sr       ; mask = 101
        lea         string,a1
        move.b      #13,d0          ; task number
        trap        #15
        movem.l     (sp)+,a1/d0     ; restore working registers
        rte

string  dc.b    'Interrupts are Cool!',0
```

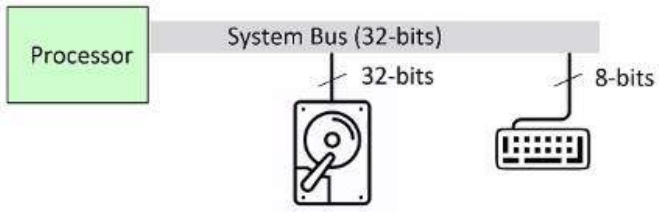
CIS2030 Microcomputers Week 11

Limitation of Polling and Interrupt-Driven I/O

- Very inefficient for data transfer to send/receive to/from memory via processor

Processor Time Allocated to Polling

- Determine the percentage of processor time required to poll each device



- Assumptions
  - Processor can execute 10 million instructions per second, and a 32-bit bus
  - Disk has a maximum transfer rate of 1 million bytes per second, and has a 32-bit bus
  - The keyboard is polled 30 times per second
  - The polling operation for each I/O device requires 20 instructions

DISK

- Transfer rate is 1 MB/s
- Transfer rate is 250 k words (32-bits) per second

- $250\,K \times 20 = 250 \times 2^{10} \times 20$

  $= 5{,}120{,}000$ instructions per second

- percentage of CPU time required for polling DISK

$$\frac{5.12 \times 10^6}{10 \times 10^6} \times 100 \cong 51\%$$

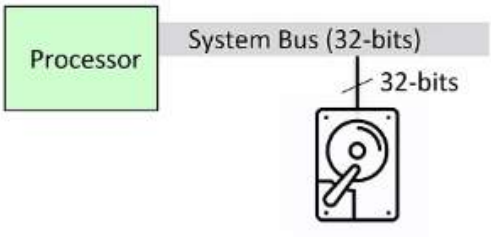
Keyboard

$20 \times 30 = 600$ instructions per second

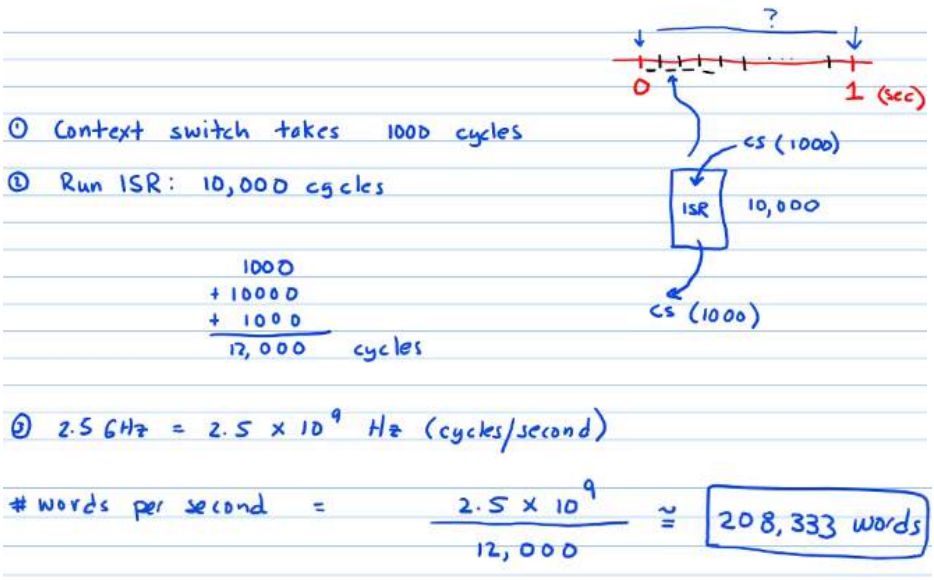
$$\frac{600}{10 \times 10^6} \times 100 = 0.006\%$$

Disk Transfer Time with Interrupts

- How many words can be transferred between the processor and the disk in one second?



- Assumptions
  - Processor has a 2.5 GHz clock
  - Context switch takes 1000 clock cycles
  - ISR takes 10000 cycles to run

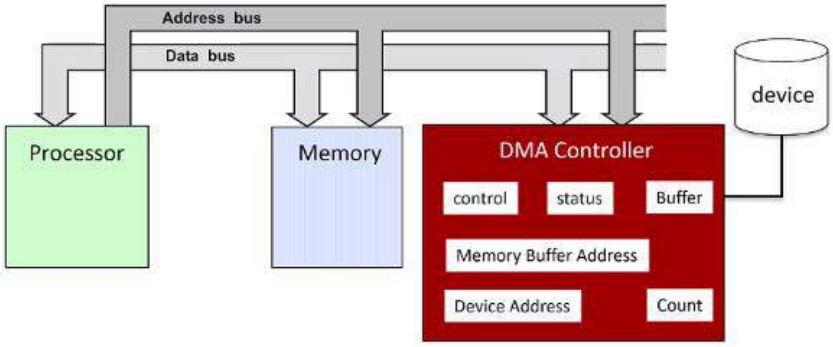


Motivation for Direct Memory Access

- Many streaming devices require large amounts of data to be transferred quickly between an I/O device and memory
    - If the processor handles the data transfer, the application may be blocked until the transfer is complete
    - Depending on the amount of data and transfer speed, the data transfer may be extremely slow

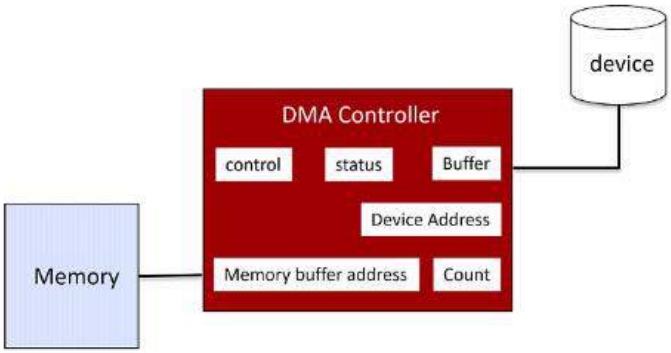
Direct Memory Access (DMA)

- Direct-Memory Access
    - Allowing the peripheral to communicate directly with the memory bypassing the CPU



DMA Controller

- All of the registers are memory-mapped and can be accessed using simple load and store operations

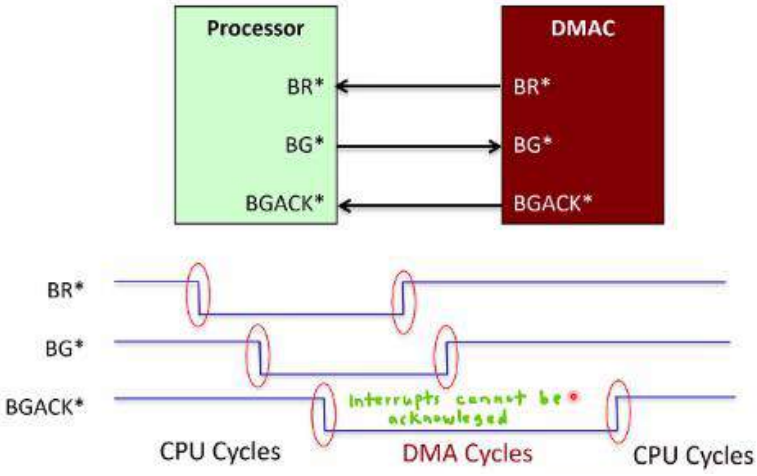


A Disk Write Based on DMA

- Transfer 256 bytes of data from a memory buffer at address 0x09000 to the disk at address 0xE4089

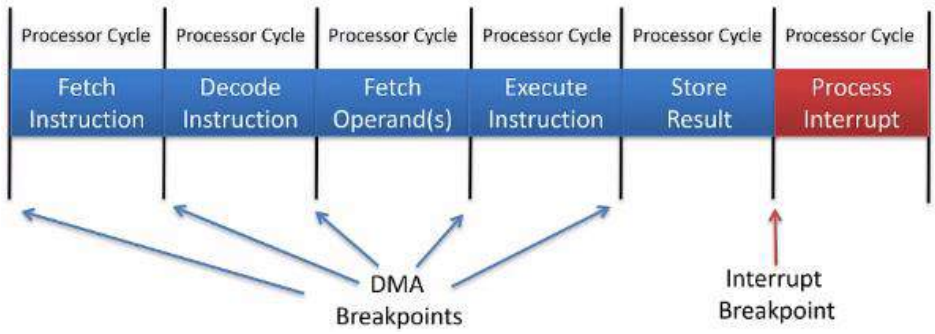
How Can the DMAC Become a Bus Master?

- The DMAC must become bus master before any transfer can occur



When Can the DMAC Become a Bus Master?

- The DMAC can become master of the bus at almost any point in the instruction cycle



TAS

TAS **Test and Set an Operand** **TAS**

Operation: Destination Tested → Condition Codes; 1 → Bit 7 of Destination

Assembler Syntax: TAS < ea >

Attributes: Size = (Byte)

Description: Tests and sets the byte operand addressed by the effective address field. The instruction tests the current value of the operand and sets the N and Z condition bits appropriately. TAS also sets the high-order bit of the operand. The operation uses a locked or read-modify-write transfer sequence. This instruction supports use of a flag or semaphore to coordinate several processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

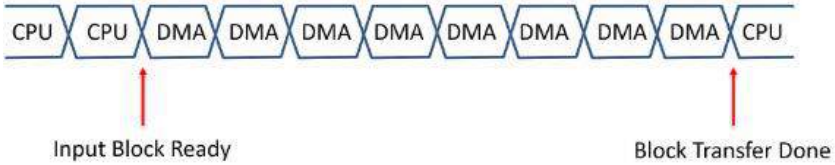
X — Not affected.
N — Set if the most significant bit of the operand is currently set; cleared otherwise.
Z — Set if the operand was zero; cleared otherwise.
V — Always cleared.
C — Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

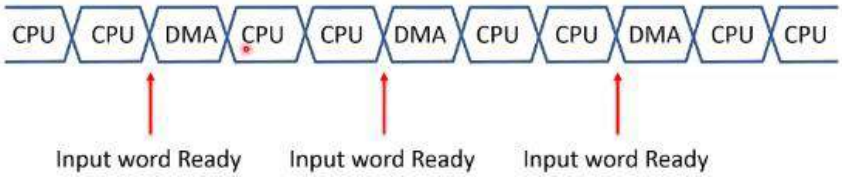
Transfer Mode: Burst

- Burst mode
  - Bandwidth of I/O = bandwidth of bus
    - Device acquires bus
    - CPU remains idle for entire duration of the transfer
    - All data (block of words) is transferred between device and memory
    - Very fast transfer, but low CPU utilization



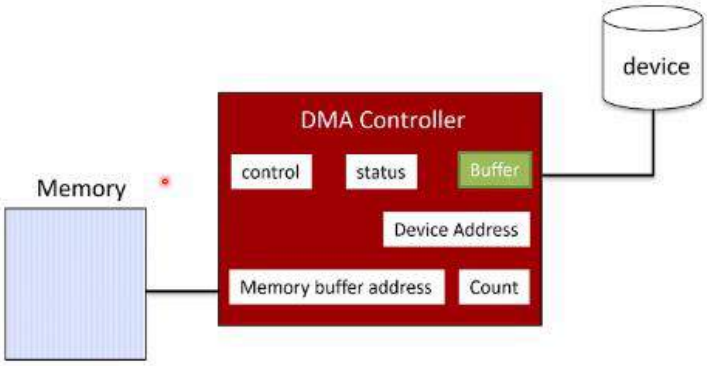
Transfer Mode: Cycle Stealing

- Cycle-stealing mode
  - Bandwidth of I/O < bandwidth of bus
    - Device acquires bus
    - CPU remains idle for entire duration of the transfer
    - One word is transferred between device and memory
    - Slower transfers, but higher CPU utilization

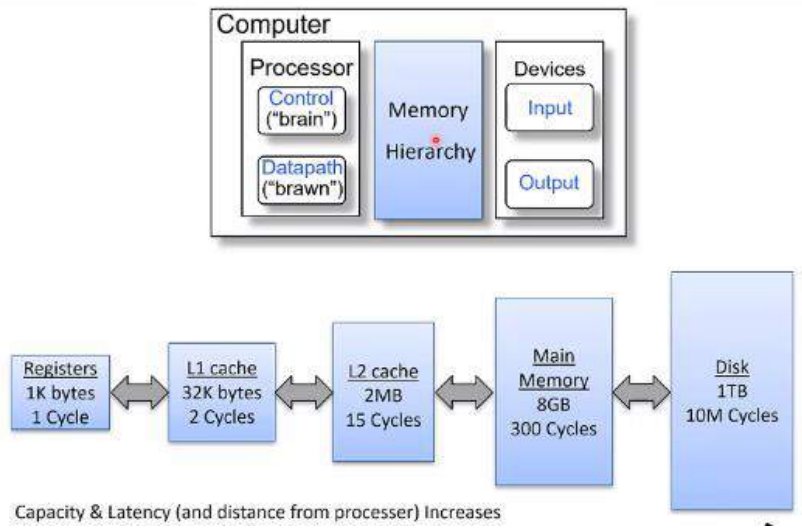


I/O Buffering

- Buffering
  - Most I/O bandwidths are less than the memory bandwidth
  - I/O buffering in the DMAC can be used to promote the more frequent use of burst mode

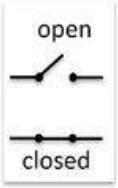


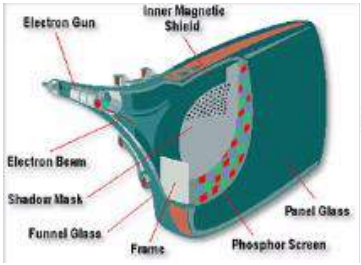
Memory Hierarchy

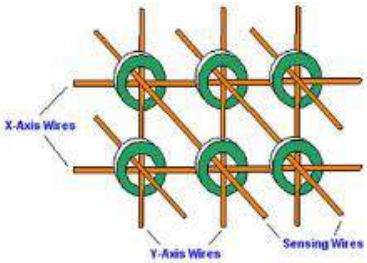


The Evolution of Memory

- Early implementations of memory were based on widely different technologies
  - Vacuum tubes (pre 1940's)
  - William's tubes (mid 1940's)
  - Magnetic core (late 1940's)



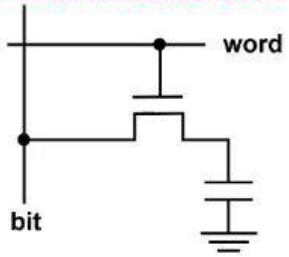




- Semiconductor memories (1970's)
  - Random-Access Memory (RAM)
  - Static RAM (SRAM)
  - Dynamic RAM (DRAM)

Dynamic Random Access Memory (DRAM)

1-Transistor DRAM Cell

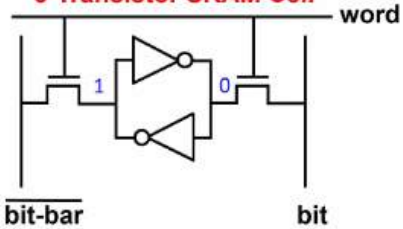


- Write:
  - 1. Drive bit line
  - 2. Select word
- Read:
  - 1. Pre-charge bit line to Vdd/2
  - 2. Select word
  - 3. Cell and bit line share charges
    - Minute voltage changes on the bit line
  - 4. Sense (fancy sense amp)
    - Can detect changes of ~10^6 electrons
  - 5. Write: restore the value
- Refresh
  - 1. Just do a dummy read to every cell

- DRAM Characteristics
  - High density, low power, cheap
  - Dynamic since data must be refreshed regularly

Static Random Access Memory (SRAM)

6-Transistor SRAM Cell



- SRAM Characteristics
  - Static since data is held without refresh
  - Lower density (1/10 of DRAM), higher cost
  - Fast access time
    - Due to location (and hence capacity), power, peripheral circuits, but not cell

Write:
- 1. Drive bit lines (bit-bar=1, bit=0)
- 2. Select word

Read:
- 1. Pre-charge bit and bit-bar to Vdd/2
- 2. Select word
- 3. Cell pulls one line low
- 4. Sense amp on column detects difference between bit and bit-bar

Location vs Access Time

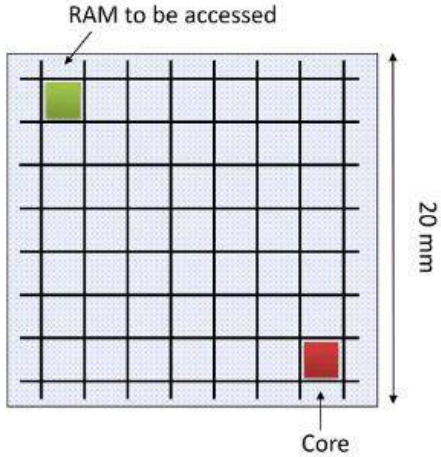
Assume 28nm technology
- Wire pitch ~ 0.1 um (microns)

DRAM Cell Area	SRAM Cell Area
2 grids -> 0.04um²	20 grids -> 0.4um²

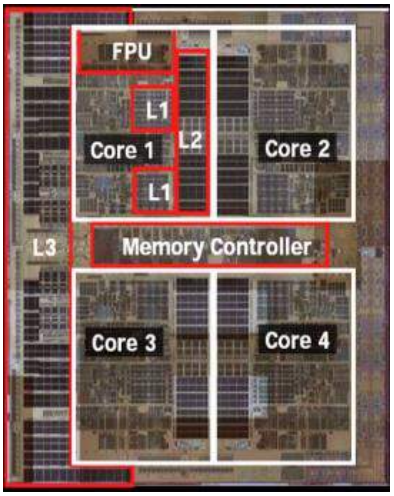
Can access a small array in a cycle

256 rows x 1024 columns

- 256K bits = 32K bytes
- 0.01mm² of DRAM
- 0.1 mm² of SRAM
- Access time: several hundred picoseconds

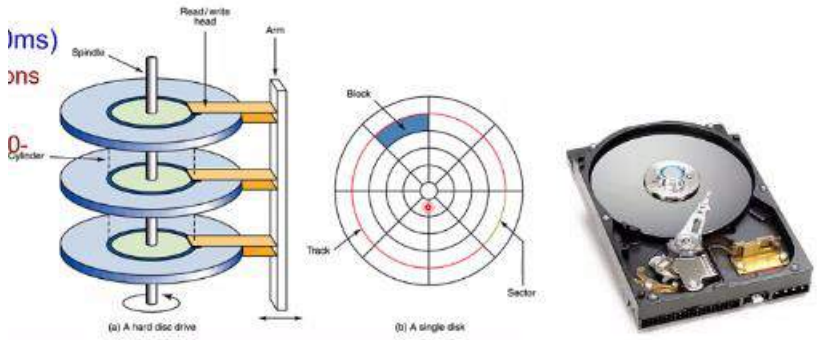


Delay approximately 2 mm/ns

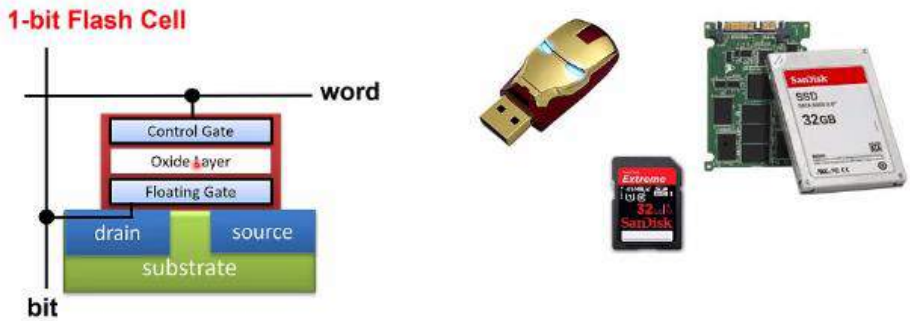


Hard (Magnetic) Disks

- Magnetic disk
  - > 1 Terabyte
- Cylindrical access
  - Seek to the require track (1-10ms)
  - Wait for the data you want to rotate under head (0-10ms)
  - Very slow access times (5-20ms(
    - That is 5-20 million instructions (or more) per access
    - Reasonable transfer rates 80-800MB/s
- Non-volatile memory
  - Holds data without power

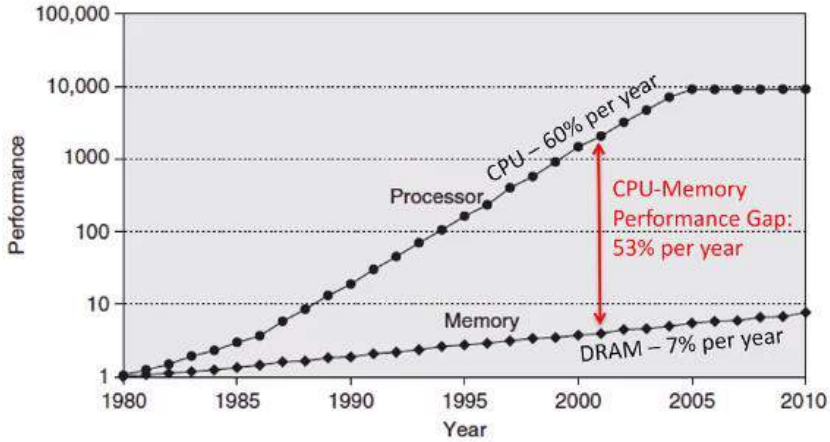


Flash Memory



- Flash memory
  - Holds contents without power (non-volatile), solid state, fast compared to disk
  - Data written in blocks, limited number of writes, slow compared to semiconductor RAM
  - Very cheap, but not as cheap as disks

Memory Wall



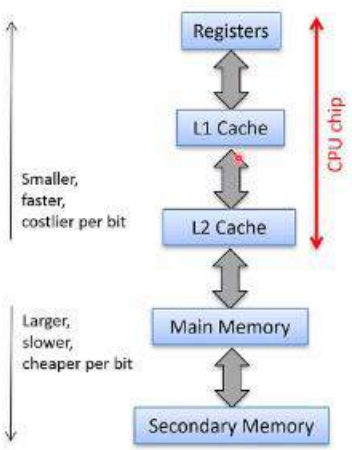
The Memory Challenge

- Goal: build a big, fast cheap memory
  - Problem: no one technology provides all three

Technology	Capacity	Latency	Cost
SRAM	100's of MBs	0.5ns – 2.5ns	$2K - $5K per GB
DRAM	100's of GBs	50ns – 70ns	$15 - $50 per GB
Magnetic Disk	10's of TBs	5ms – 20ms	$0.05 - $1 per GB

- In general:
  - Big memories are slow
    - Even when build using the fastest components
  - Fast memories are expensive

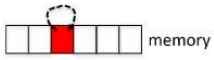
Memory Hierarchy



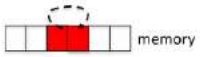
Access Time	Capacity	Technology
1 cycle	~500 Bytes	SRAM
1-3 cycles	~64KB	SRAM
10-20 cycles SRAM	1-100MB	
~100 cycles	~10GB	DRAM
10^6-10^7 cycles	~100G-10TB	Disk/Flash

Locality

- Principle of Locality
  - Programs tend to use instructions and data with addresses that are nearby (or equal) to those they have used recently
- Temporal Locality
  - Recently referenced items are likely to be referenced again in the near future

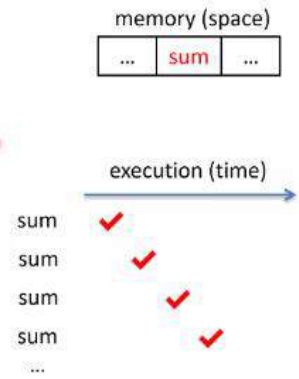


- Spatial Locality
  - Items with nearby addresses tend to be referenced close together in time



Find the Locality

```
for(i=0; i<SIZE; i++)
    sum = sum + a[i];
```

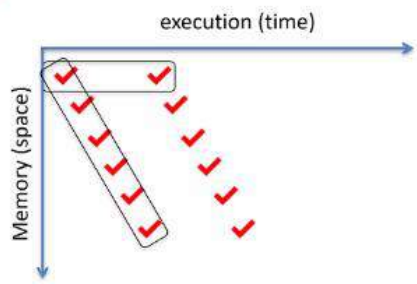


- Data
  - Spatial: array `a[i]`
    - accessed in stride-1 pattern
    - exploit by fetching nearby data into higher levels of hierarchy
  - Temporal: `sum`
    - Sum referenced in each iteration
    - Exploit by keeping recently accessed data in higher levels of hierarchy

```
for(i=0; i<SIZE; i++)
    sum = sum + a[i];
```

- Instructions
  - Spatial: reference instructions in sequence
  - Temporal: cycle through loop repeatedly
    - both can benefit from fetching loop body into higher levels of hierarchy

```
      ...
LOOP  CMPI  #SIZE,D0
      BGE   EXIT
      ADD   (A0,D1),D2
      ADD    #2,D1
      ADD    #1,D0
      BRA    LOOP
EXIT  ...
```



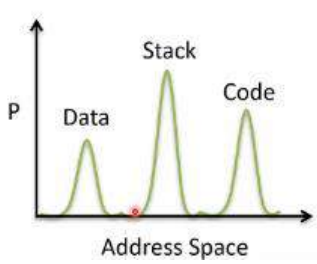
Probability Density Functions

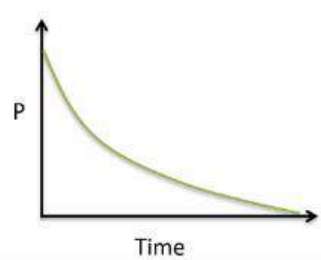
- Spatial Locality
  - Likely to reference items near recent references

```
for(i=0; i<SIZE; i++)
    sum = sum + a[i];
```

- Temporal Locality
  - Likely to reference item that was recently referenced

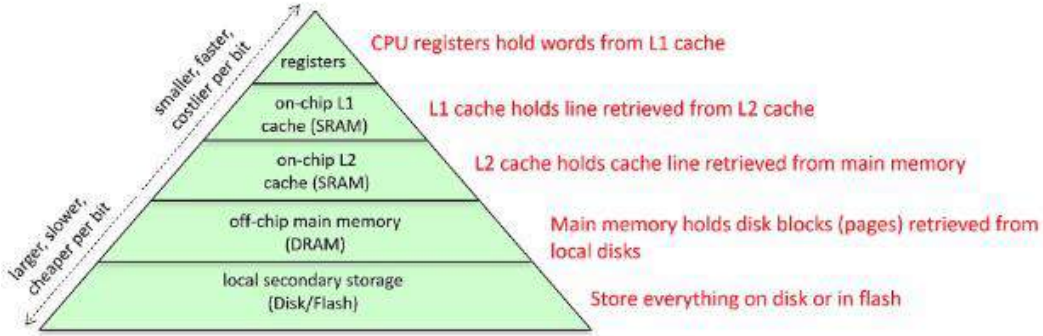
```
a = p.x + p.y
b = p.y - p.x
c = p.x * p.x - p.y * p.y
```



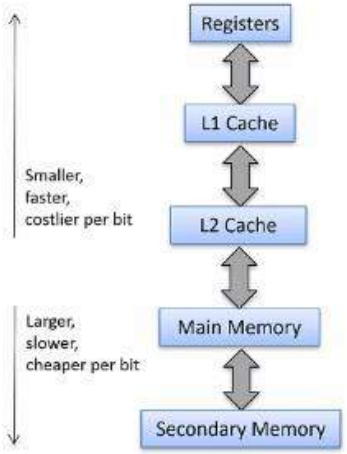


Taking Advantage of Locality

- Fundamental ideas of memory hierarchy
  - Each level K serves as a cache for the larger, slower, cheaper level, K+1 below
  - Because of locality, programs tend to access data at level K more often than at level K+1



Who Manages Each Cache Memory?



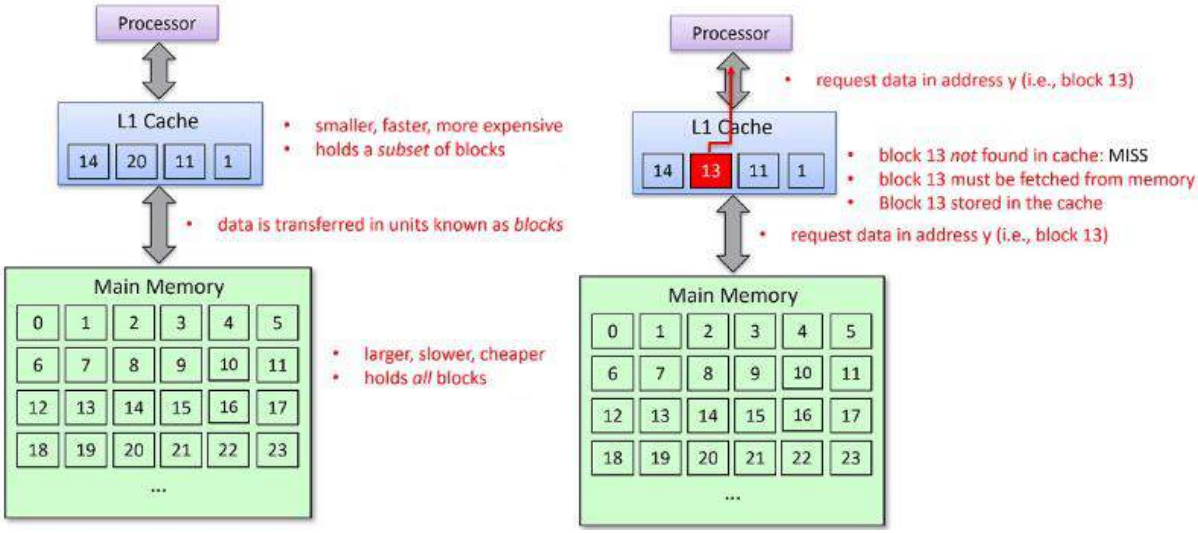
	Access Time	Capacity	Managed by
Registers	1 cycle	~500 Bytes	software/compiler
L1 Cache	1-3 cycles	~64KB	hardware
L2 Cache	10-20 cycles	1-100MB	hardware
Main Memory	~100 cycles	~10GB	software/OS
Secondary Memory	10^6-10^7 cycles	~100G-10TB	software/OS

Cache Basics

- General definition:
  - A hidden storage space for provisions, weapons, and/or treasures
- Computer science definition:
  - A smaller, faster, cheaper memory used for the storage of frequently or recently used data and instructions
- Computers often use multiple caches
  - Level-1 data-cache and instruction-cache
  - Web caches used on the internet

- Disk block caches
- Browser caches pages

Basic Ideas Illustrated with Single L1 Cache

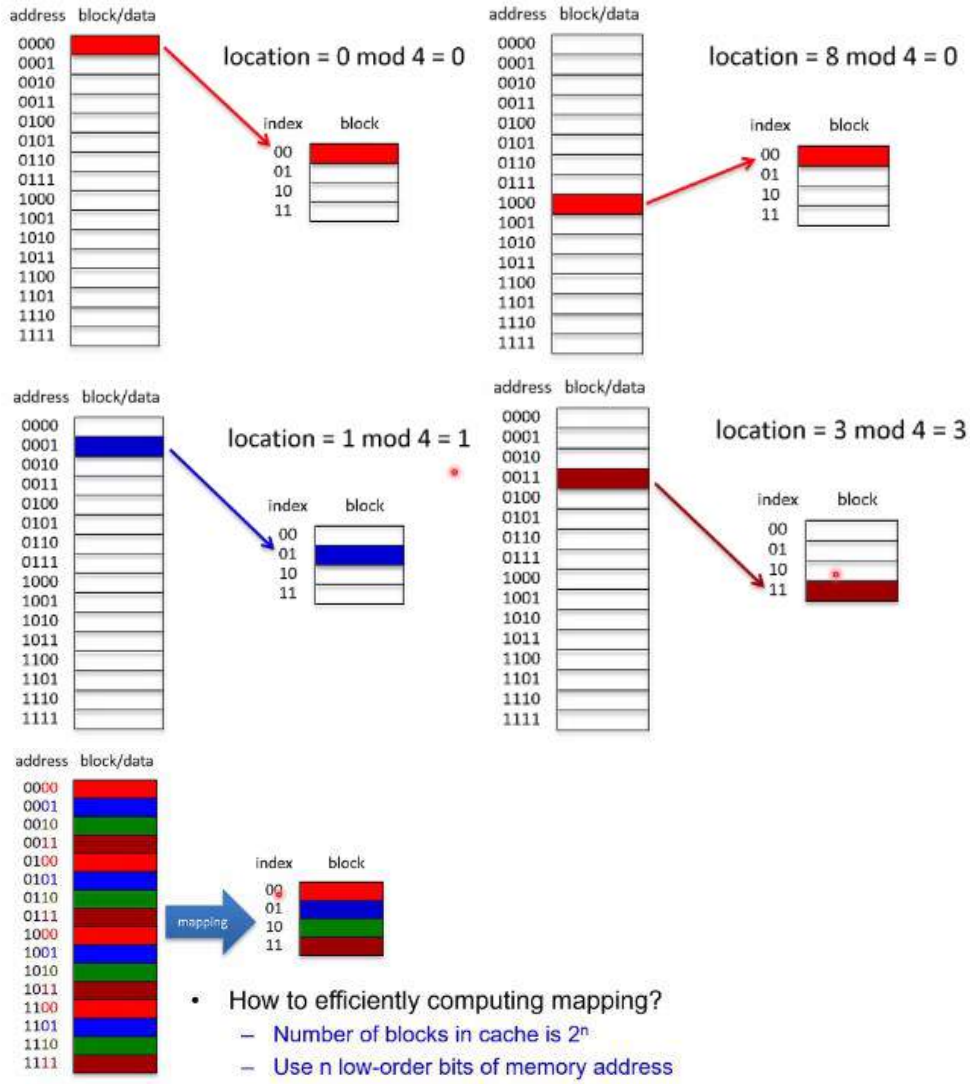


Questions Related to Caches

- Where can a particular block be stored?
  - Mapping policy
    - Directed-mapped cache (single location)
    - Fully-associative cache (anywhere)
    - Set-associative cache (anywhere)
- What block gets thrown out to make room?
  - Replacement policy
- How much data do I move?
  - Block size
- How do we handle writes?
  - Bypass cache
  - Write thru cache
  - Write into cache - then write back to memory

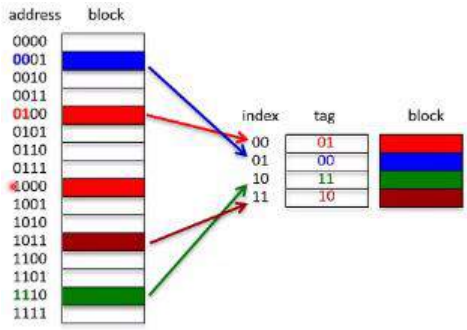
Direct-Mapped Cache

- Main-memory (block) address is used to determine location in cache
- Direct-mapped caches: only one choice
  - Cache location = (Memory Address of block) modulo (Number of blocks in Cache)



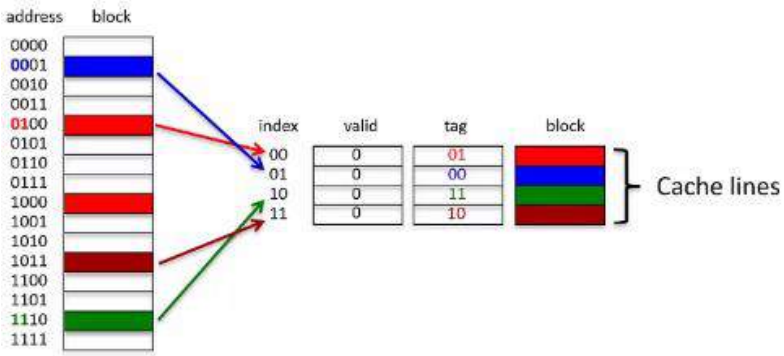
Tag Bits

- How do we know which particular block is stored in a cache location?
  - Store the block address as well as the data
  - Only need to use the upper bits of the memory address
  - Called the tag



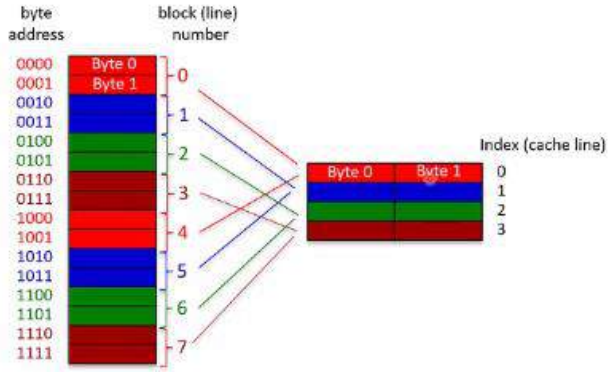
Valid Bits

- How do we know if there is no data in a location in the cache?
  - Valid bit
    - 1 = present, 0 = not present
  - Initially set to zero (called flushing the cache)

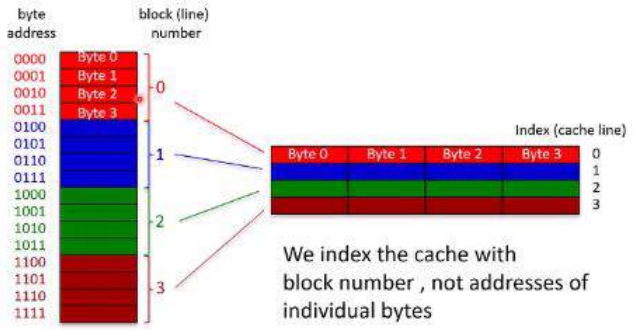


Cache Blocks (Size)

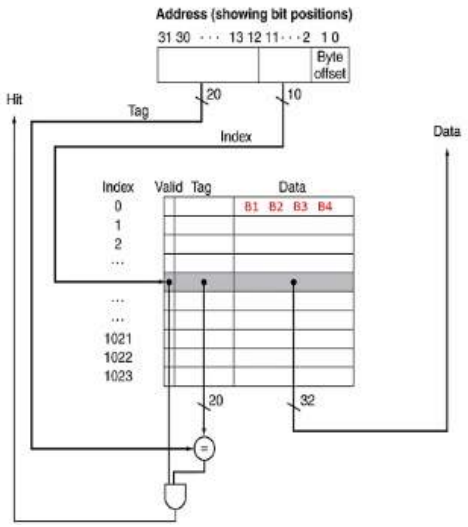
- Purpose: exploit spatial locality and amortize overheads
  - Assume a block size of 2 bytes



  - Now assume a block size of 4 bytes
  - Typical cache line size: 32/64 bytes



Direct Cache Organization



- Assumptions
  - 32-bit memory address
  - 4K = 4 x 2^{10} = 4096 byte cache
  - 1K = 2^{10} = 1024 (4-byte) blocks/cache lines
- Steps
  1. Use index to read V, tag from cache
  2. Compare read tag with tag from address
  3. If match, return data & hit signal
  4. Otherwise, return miss

Direct Cache Example

- Memory Assumptions
  - 7 bit (byte) address
  - 32 words (4 bytes each)
- Cache Assumptions
  - 8 blocks
  - 1 word/block

Word Address	Binary Address	Hit/ Miss	Cache Line
18	10010	MISS	010

Index	V	Tag	Data (Block)
000	1	10	memory[10000]
001	0		
010	1	10	memory[10010]
011	1	00	memory[00011]
100	0		
101	0		
110	1	10	memory[10110]
111	0		

- Cache Requests
  - Word 22
  - Word 26
  - Word 22
  - Word 16
  - Word 3
  - Word 16
  - Word 18 (replacement)

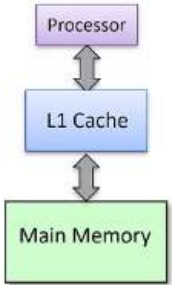
CIS2030 Microcomputers Week 12

Cache Terminology

- **Block**: unit of information present at any cache level
  - Also known as a cache line
  - Typical block size: 32 or 64 bytes
- **Hit**: data (block) found in cache
- **Hit rate**: fraction of memory accesses that hit
  - Typical numbers for L1 cache: 90% to 95%
  - What about L2?
- **Hit time**: time to deliver a line in the cache to the processor
  - Includes time to determine if the line is in the cache
  - **Typical hit time**: 1-3 clock cycles for L1 cache, higher for other levels
- **Miss**: data (block) is not found in cache
  - Huge difference between miss and hit times
  - Could be ~100x, if just main memory and L1 cache
- **Miss rate**: fraction of memory accesses that do not hit
  - **Miss rate** = 1 - hit rate
  - Typical numbers for L1 cache: 5% to 10%
- **Miss penalty**: additional time required because of miss
  - **Miss penalty** = higher level access time + time to deliver to lower level + cache replacement / forwarded to processor time
  - Miss penalty is usually much larger than hit time
  - This is in addition to hit time

Average Memory Access Time

- Average Memory Access Time (AMAT): the average access time that the processor sees for a memory reference
    - AMAT = hit time + miss rate x miss penalty
- Assume
    - Hit rate of 97%
        - Miss rate of 3%
    - Cache hit time of 1 cycle
    - Cache miss penalty of 100 cycles
- Average Access time?
    - 1 cycle + 0.03 x 100 = 4 cycles
    - 1 cycle + 0.01 x 100 = 2 cycles if hitt rate is 99% and miss 1%



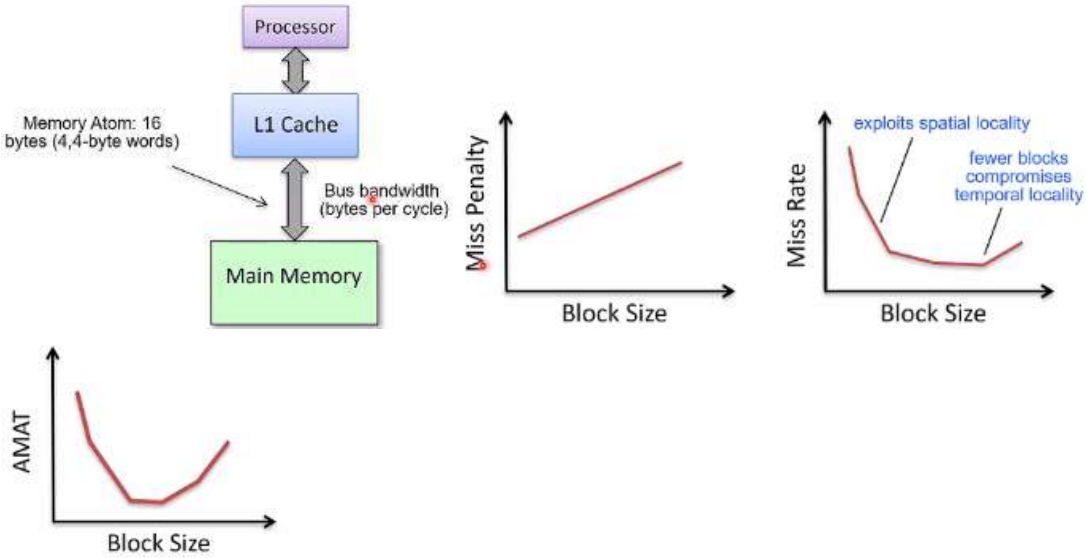
- Focus on minimizing miss rate

Why Do We Miss in the Cache?

- Compulsory miss
    - Also known as “cold” miss
    - Occurs first time a block is referenced
        - Words 22, 26, 16 and 3 in earlier example
    - Can reduce through hardware prefetching and increasing block size
- Capacity miss
    - Occurs when there is not enough room in the cache to hold all of the active blocks (also known as the working set)
    - Can reduce by making cache bigger
- Conflict miss
    - Occurs when cache is large enough, but multiple blocks map to the same location (or set of locations)
    - Can be reduced by increasing the associativity
        - Words 18 and 26 in earlier example

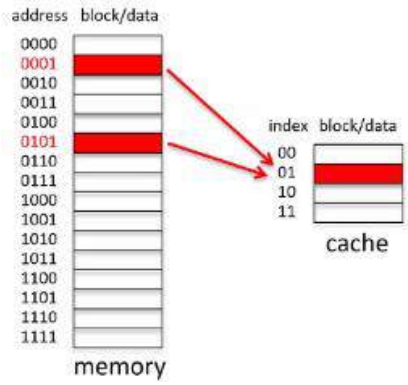
Cache Misses and Increased Block Size

- If larger block sizes take advantage of spatial locality, why limit ourselves to 32 or 64 bytes?
    - Miss penalty increases, as it takes longer to transfer large blocks
    - Large blocks can increase the miss rate
    - Large blocks can increase AMAT
        - AMAT = Hit Time + (Miss Rate x Miss Penalty)
    - No one block size is “best”



Problems with Direct-Mapped Caches

- Problem 1: Conflict Misses
    - Two blocks used concurrently map to the same location
    - Only one can fit in cache, regardless of cache size
- Program references addresses
    - 1, 5, 1, 5, …
- Problem 2: Thrashing
    - If alternating accesses, blocks are evicted before reuse
    - Nullifies benefit of caching
- Conflicts and thrashing happen frequently



The Problem

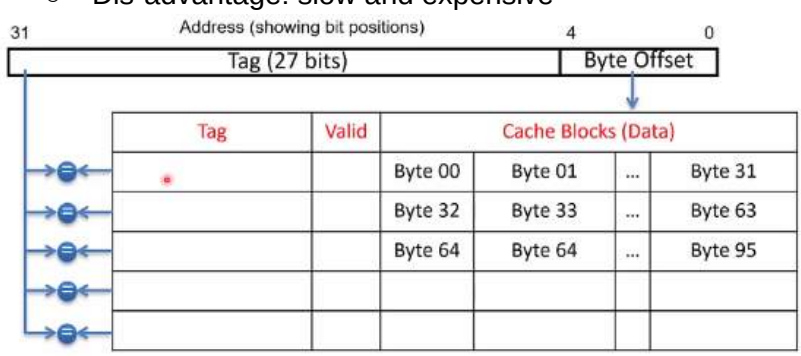
- Assume that we have a small 64K byte cache
- Consider the following code:

```
#define size 8192

double a[size], b[size], c[size];
...

for (i=0; i < size; i++)
        c[i] = a[i] + b[i];
```

- What happens?
    - Cache misses occur with every array access (i.e., 3 per iteration)
    - Spatial locality savings will be nullified
- How can conflicts be reduced?

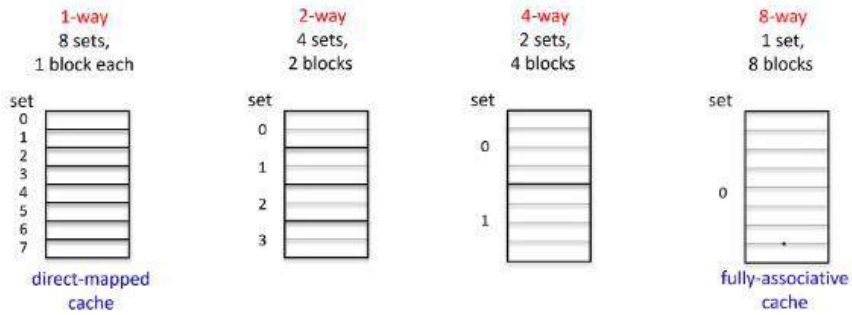
Fully Associative Cache

- Store a block anywhere within the cache
    - No cache index, but compare tag to all cache entries
    - Advantage: no conflict misses, only capacity
    - Dis-advantage: slow and expensive



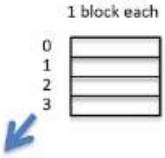
N-Way Set-Associative Cache

- Compromise between directed-mapped and fully-associative cache
  - A cache contains some number of sets
  - Each set stores (up to) N blocks
  - Helps to avoid conflict misses, and fast because all blocks in a set are searched in parallel



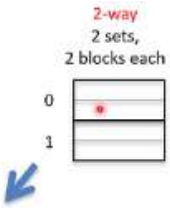
Cache Comparison

- Directed mapped, 2-way set associative, fully associative
  - Assume 4-block cache
  - Block access sequence: 0, 8, 0, 6, 8
- Directed mapped
  - (Memory address) mod (number of blocks in cache)



Memory Address	Cache Block	Hit/Miss	Cache Set 0	Cache Set 1	Cache Set 2	Cache Set 3
0	0 mod 4 = 0	miss	mem[0]			
8	8 mod 4 = 0	miss	mem[8]			
0	0 mod 4 = 0	miss	mem[0]			
6	6 mod 4 = 2	miss	mem[0]		mem[6]	
8	8 mod 4 = 0	miss	mem[8]		mem[6]	

- 2-way set associative
  - (Memory address) mod (number of sets in cache)



Memory Address	Cache Block	Hit/Miss	Cache Set 0 (Even Address)		Cache Set 1 (Odd Address)	
0	0 mod 2 = 0	miss	mem[0]			
8	8 mod 2 = 0	miss	mem[0]	mem[8]		
0	0 mod 2 = 0	hit	mem[0]	mem[8]		
6	6 mod 2 = 0	miss	mem[0]	mem[6]		
8	8 mod 2 = 0	miss	mem[8]	mem[6]		

- Fully associative
  - 1 set, any memory address can be stored in any cache block

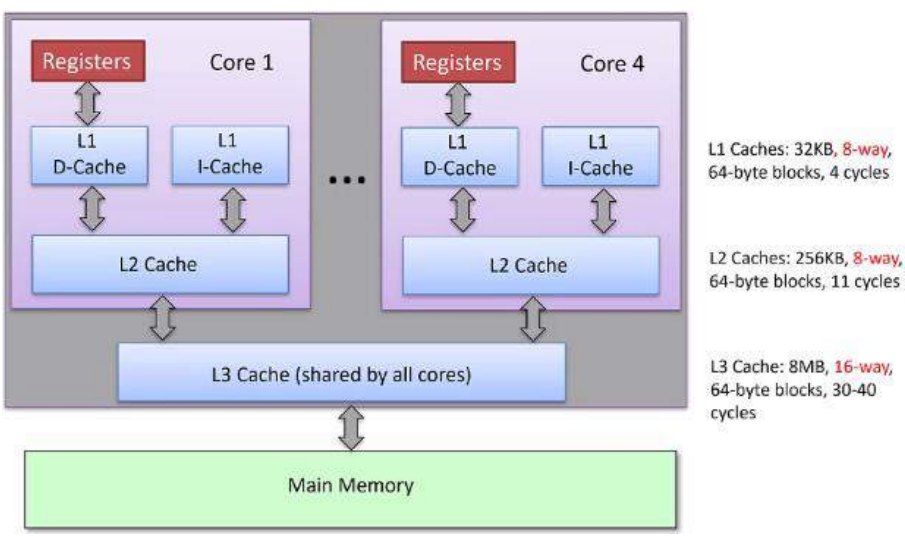
Memory Address	[redacted]	Hit/Miss	Cache Contents			
0	[redacted]	miss	mem[0]			
8	[redacted]	miss	mem[0]	mem[8]		
0	[redacted]	hit	mem[0]	mem[8]		
6	[redacted]	miss	mem[0]	mem[8]	mem[6]	
8	[redacted]	hit	mem[0]	mem[8]	mem[6]	

Misses and Associativity in Caches

- Miss rates decrease as associativity increases
  - Due to reduction in conflict misses
  - Caveat: diminishing returns
- Simulation results for 10 SPEC2000 programs on processor with 64K bytes of L1 D-Cache, 48-byte blocks

Associativity	Data Miss Rate
1-way	10.3%
2-way	8.6%
4-way	8.3%
8-way	8.1%

Cache Hierarchy for Intel Core i7



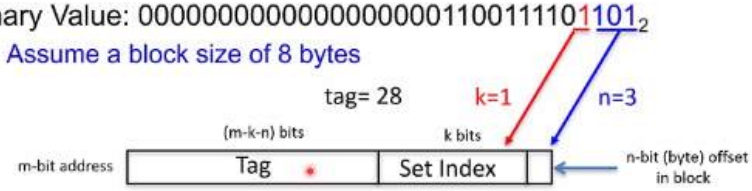
How to Locate Data in a Set-Associative Cache?

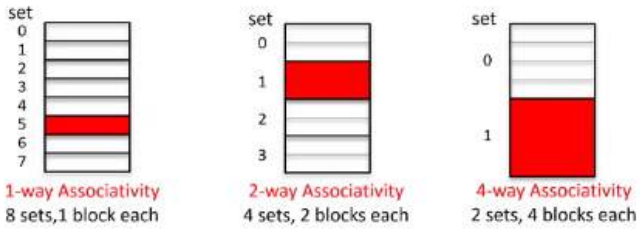
- Before

- Now

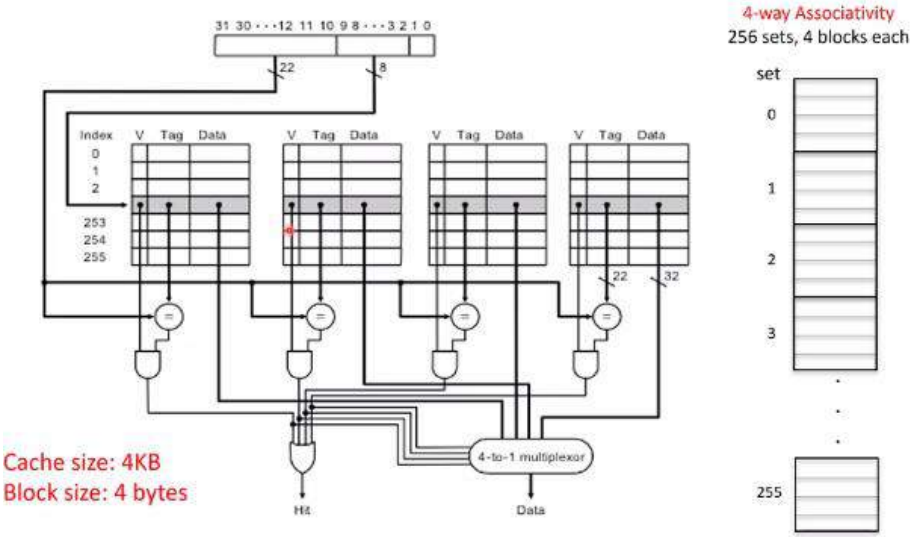
Example

- Consider the following address: 0x19ED
- Binary Value: 0000000000000000000110011110110 1 101$_2$
  - Assume a block size of 8 bytes



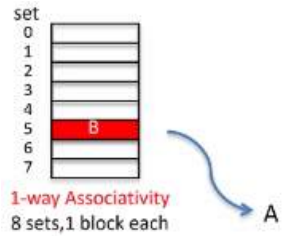


4-Way Set-Associative Cache Implementation

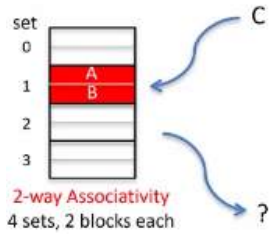


Block Replacement Policy

- In the case of a conflict miss, which block gets replaced?
- Directed mapped
  - Easy, only one choice
  - Replace the cache line at the index you need



- N-way set-associative cache
  - Must choose which “way” to replace
  - Random (became very unpopular)
  - Least Recently Used (LRU)
    - Replace block that has been unused for the longest time
    - Promotes locality



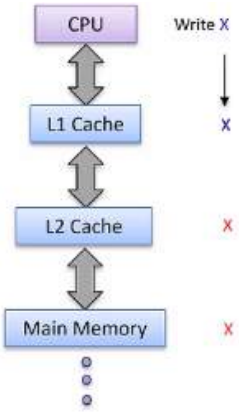
Least-Recently Used Implementation

- Need to keep track of ordering
  - For N-ways there are N-factorial orderings
    - 2-way: 2 possible orderings
    - 3-way: 6 possible orderings
    - Use update table to keep track of orderings

Current State	Order (most -> least recent)	Way 0	Way 1	Way 2
0	0, 1, 2	0	2	4
1	0, 2, 1	1	2	4
2	1, 0, 2	0	2	5
3	1, 2, 0	0	3	5
4	2, 0, 1	1	3	4
5	2, 1, 0	1	3	5

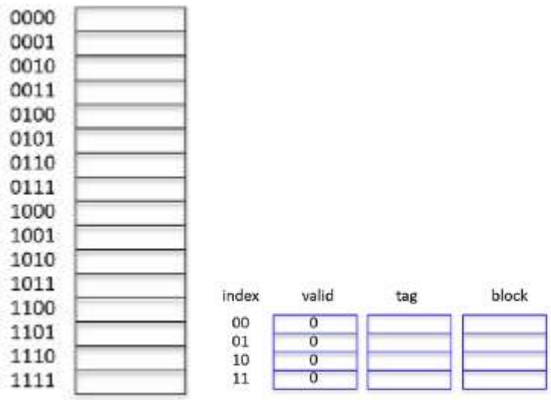
Cache Writes

- Observation
  - Multiple copies of a cache block may exist
    - L1, L2, L3, Main memory, etc.
- Problem
  - Writes can leave the memory in an inconsistent state
  - Where do we put the data we write?
    - In the cache?
    - In main memory?
    - In both?



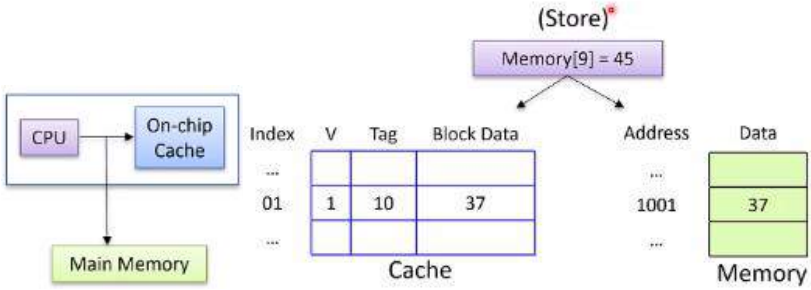
Working Assumptions

- Memory
  - 16 words
- Cache
  - Directed mapped, 4-blocks (1 word per block)
  - Assume writes always HIT



Write-Through Cache

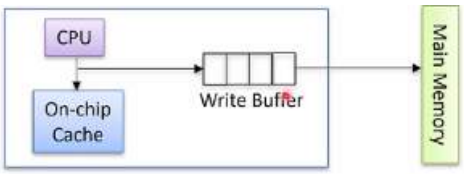
- Write-through
  - Main memory is updated on each cache write



- Pros
  - Simple
  - Maintains memory consistency
- Cons
  - Writes can cause significant delay if processor must stall

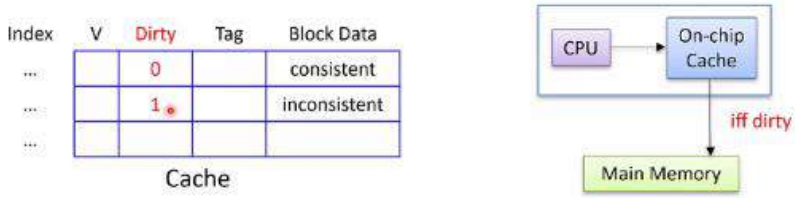
Write-Buffer

- Write buffer between cache and memory
  - Processor writes data into cache and write buffer
  - Memory controller slowly “drains” buffer to memory
- Write Buffer
  - First-in-First-Out (FIFO)
  - Holds a small number of writes
  - Can absorb small bursts from CPU as long as the long-term rate of writing to the buffer does not exceed the maximum rate of writing to DRAM

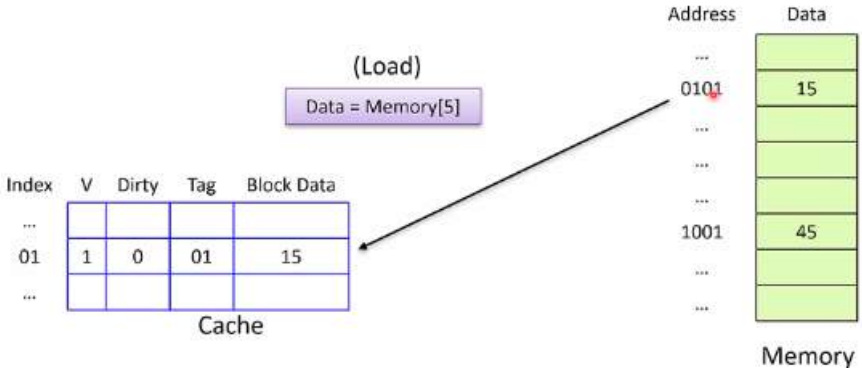


Write-Back Cache

- Write-Back
  - Only the cache entry is updated on each cache write so main memory and cache data are (temporarily) inconsistent
    - Add “dirty” bit to the cache entry to indicate whether the data in the cache entry must be committed to memory
  - Replacing a cache entry requires first writing the block back to memory before replacing the entry if it is “dirty

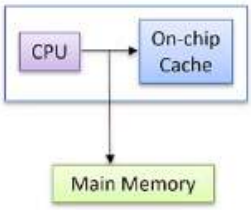


Example

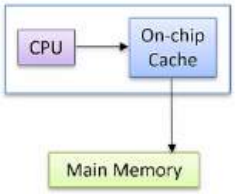


Write Policy Pros and Cons

- Write through (used sometimes)
  - Easier to implement
  - Requires a lot of bandwidth to the main memory
    - Write-buffer employed



- Write-back (most common)
  - Writes are fast (on a hit)
    - No need to write to main memory
  - Multiple writes to a single address (temporal locality) or within a block (spatial locality) only require a single write-back later
  - Cost of write-back is 2 memory accesses
    - Write buffer can be used here, too

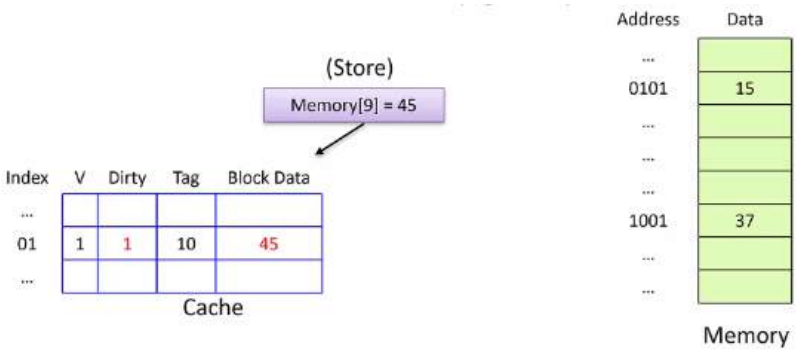


What To Do On a Write Miss?

- What happens on a cache miss?
  - Two questions
- Do we allocate space in the cache for the address?
  - Write-allocate (bring block in)
  - No-write allocate (write to main memory)
- Do we fetch the rest of the block contents from memory?
  - Arises when you do a write-allocate
  - Store updates 1-word from a larger block of words
  - Fetch-on-miss vs no-fetch on miss
    - For no-fetch-on-miss must remember which bits are valid
    - Use additional valid bits in each cache line

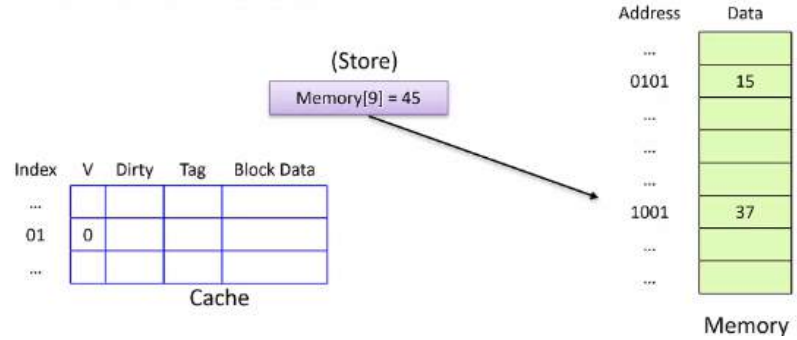
Write-Allocate (Example)

- Write-Allocate
  - Load data into cache, update line in cache
    - Good if more writes to data soon follow (e.g., stack)



No-Write-Allocate (Example)

- No Write-Allocate
  - Write directly to memory
  - Also known as “write-around”



Write-Allocate or No-Write Allocate?

```
for(i=0; i<HUGE_SIZE; i++)
    a[i] = i;
```

- Observation
  - Code is producing a “huge” stream of data
  - If we write-allocate, this stream will eventually knock everything out of the cache
    - Also known as “cache pollution”
  - When you are not going to use data soon it may be better to write around the cache to main memory
    - Non-temporal stores

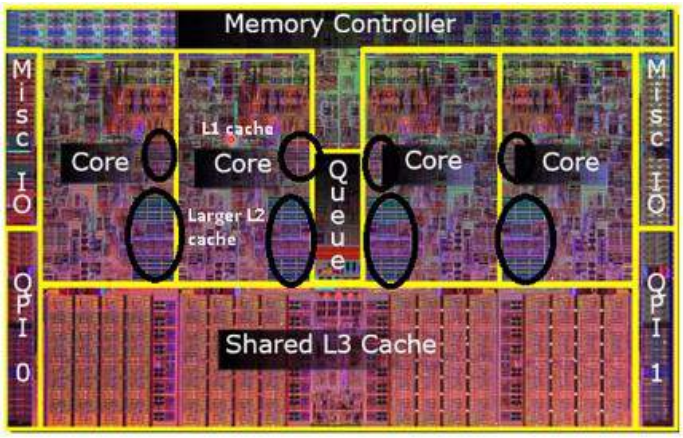
Typical Cache Configurations

- Commodity hardware supports multiple cache policies
  - Selected by Operating System
  - Typically applied to an individual page (e.g., 4K bytes)
- Typical Cache configurations

Write-back Caches	Write-through Caches

• Write-allocate, fetch-on-miss
  • most common
	• Write-allocate, fetch-on-miss
	  • spatial locality
	• Write-allocate, no-fetch-on-miss
	  • streaming pattern, but fits in cache
	• No-write-allocate
	  • streaming pattern, but does not fit in cache

Core i7 (First Nehalem Processor)



Per Core

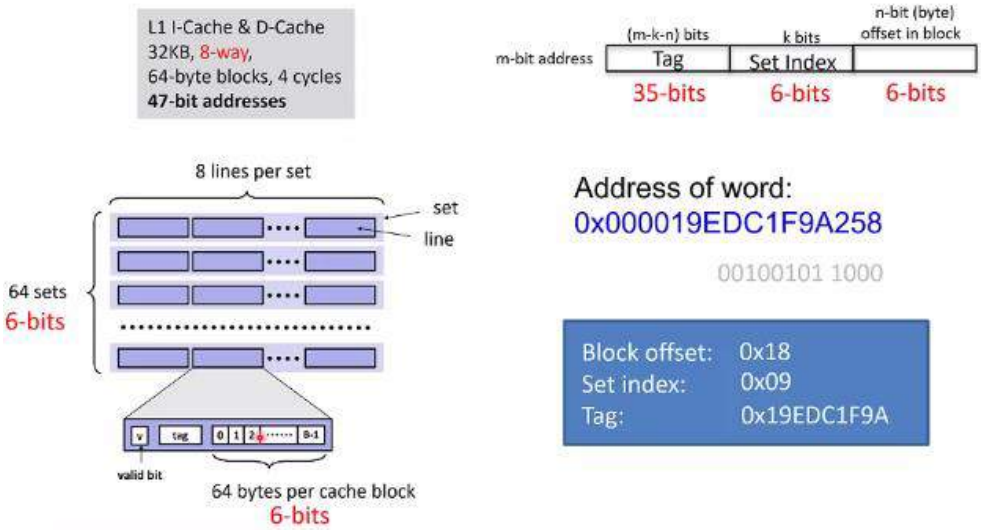
L1 I-Cache & D-Cache 32KB, 8-way, 64-byte blocks, 4 cycles

L2 Unified Cache 256KB, 8-way, 64-byte blocks, 11 cycles

Shared

L3 Caches: 8MB, 16-way, 64-byte blocks, 30-40 cycles

Core i7: L1-Data Cache



Tips for Writing Cache-Friendly Code

- Cache memories can have a significant impact on code performance
- Coding tips
  - Focus on the inner loops, were the bulk of computations and memory accesses occur
  - Minimize the misses in inner loop
    - Once an item is read from memory, try to use it frequently
      - Good for temporal locality
    - Try to read data items sequentially from memory with a stride of 1
      - Good for spatial locality

Example: Summing a 1D Array

```
int sumarray(int *array, int size)
{

        int i,sum=0;

        for(i=0;i<size;i++)
            sum += array[i];
        return sum;
}
```

Assumptions:
- `int` (4 bytes)
- `array` is block aligned
- Cold cache
- block size is 4 words
- 1 word is 4 bytes

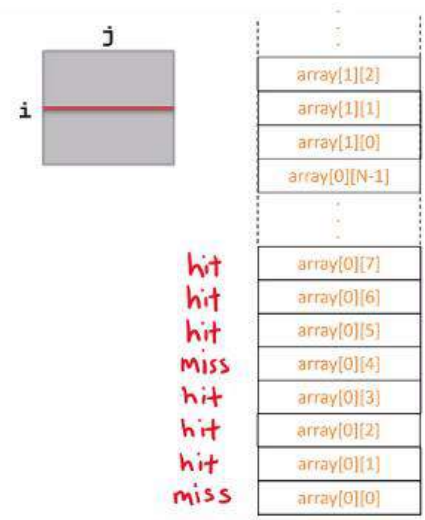
array[i]	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
Access order, hit or miss	1,miss	2,hit	3,hit	4,hit	5,miss	6,hit	7,hit	8,hit

Example: Summing a 2D Array by Row

```
int sumrows(int array[M][N]) {

    int i,j,sum=0;

    for(i=0;i<M;i++)
      for(j=0;j<N;j++)
          sum += array[i][j];
    return sum;
}
```

Miss Rate: 0.25

Assumptions:
- `int` (4 bytes)
- `array` is block aligned
- C stores arrays in row-major order
- block size is 4 words
- N>>4, and 4MN > cache size



- Stepping through columns in one row
  - Accesses successive elements
  - If block size > sizeof(array[i][j]) bytes, exploits spatial locality
    - Miss rate = sizeof(array[i][j]) / block size

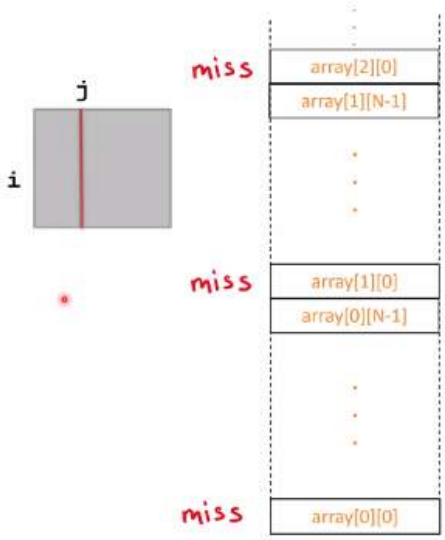
Example: Summing a 2D Array by Column

```
int sumcols(int array[M][N]) {

    int i,j,sum=0;

    for(j=0;j<N;j++)
      for(i=0;i<M;i++)
          sum += array[i][j];
    return sum;
}
```

Miss Rate: 1.0

Assumptions:

- `int` (4 bytes)
- `array` is block aligned
- C stores arrays in row-major order
- block size is 4 words
- N>>4, and 4MN > cache size

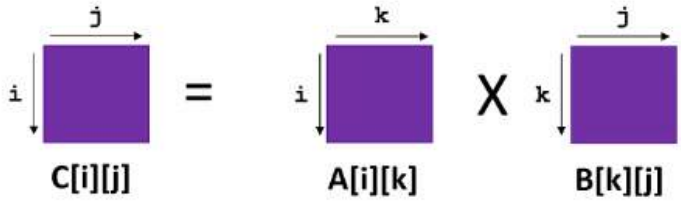


Example: Multiplying 2 Matrices

```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    sum=0;
    for(k=0; k<n; k++)
      sum += a[i][k] * b[k][j]
    c[i][j] = sum;
  }
}
```

Description

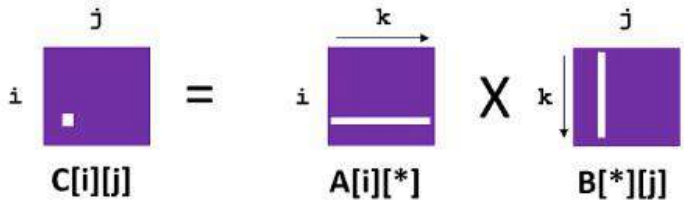
- N x N array multiplication
- Matrix elements are doubles (8 bytes)
- $O(n^3)$ total operations
- Accesses:
  - N reads per source element
  - N values summed per destination



Example: Multiplying 2 Matrices (ijk)

```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    sum=0;
    for(k=0; k<n; k++)
      sum += a[i][k] * b[k][j]
    c[i][j] = sum;
  }
}
```

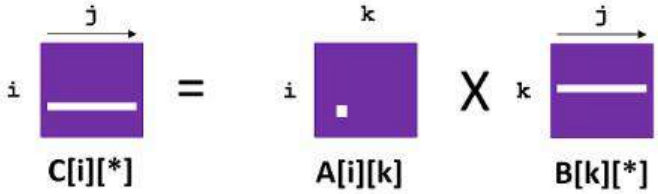
Inner Loop Miss Rates		
A	B	C
0.25	1.0	0.0



Example: Multiplying 2 Matrices (kij)

```
for(k=0; k<n; k++){
  for(i=0; i<n; i++){
    r = a[i][k];
    for(j=0; j<n; j++)
      c[i][j]= r * b[k][j]
    }
}
```

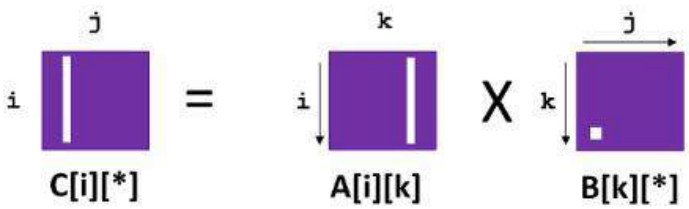
Inner Loop Miss Rates		
A	B	C
0.0	0.25	0.25



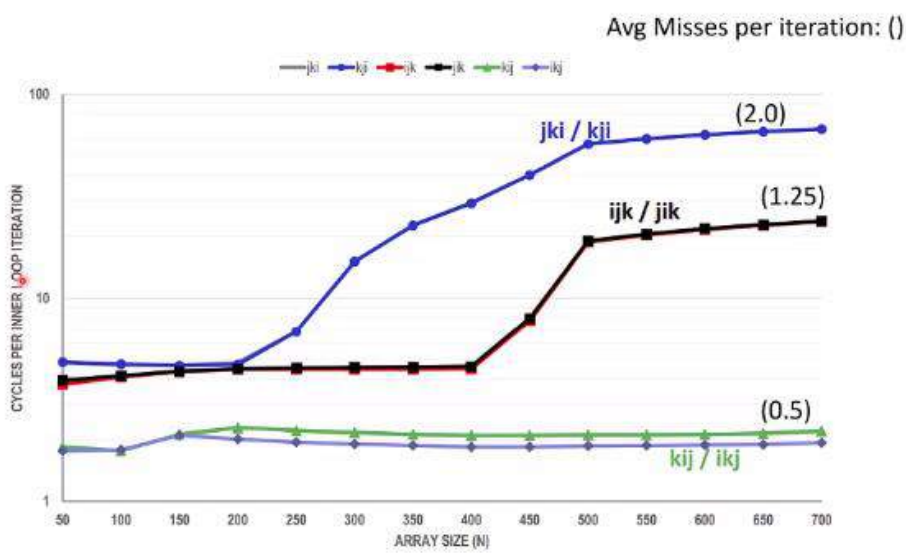
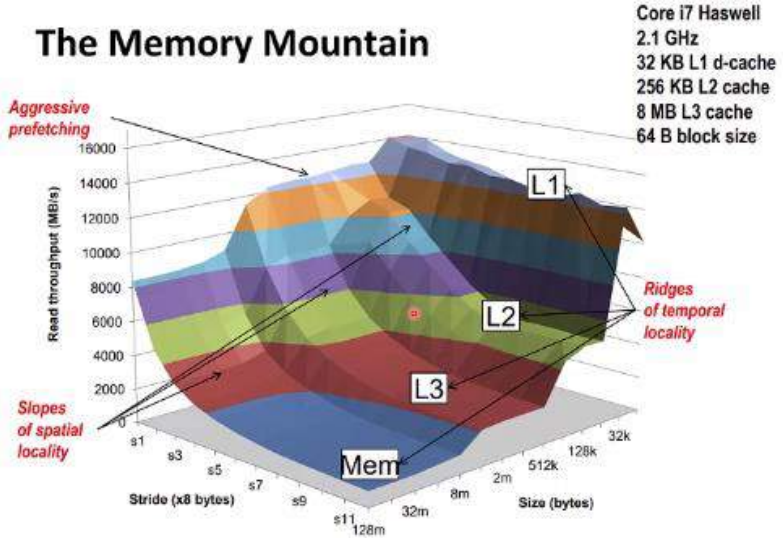
Example: Multiplying 2 Matrices (jki)

```
for(j=0; j<n; j++){
  for(k=0; k<n; k++){
    r = b[k][j];
    for(i=0; i<n; i++)
      c[i][j]= a[i][k] * r;
    }
}
```

Inner Loop Miss Rates		
A	B	C
1.0	0.0	1.0



Performance Comparison



Cache Capabilities (Core i7)