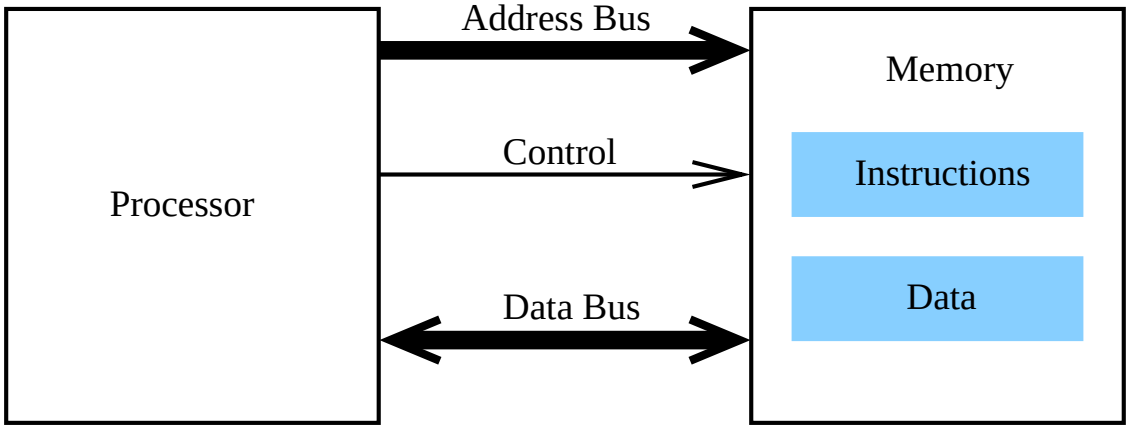
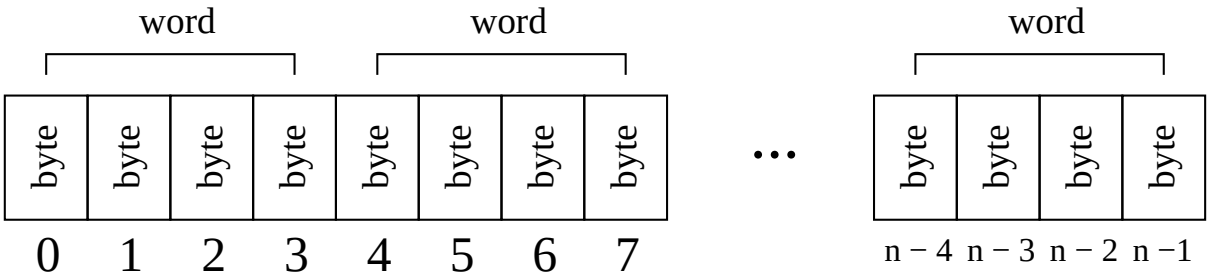
von Neumann machines

The *von Neumann machine* is the basis of most (but not all) modern computers. Another common name is *stored program machines*.



Memory Organization: Array of bytes



Two key characteristics are:

1. Common memory (i.e., storage) for instructions and data
2. Use of a program counter

Common memory

- Memory is organized into *bytes* (8 bits) of data (compare to a 1-D array)
- Memory is referenced by an *address* (e.g., a number)

Why a common memory?

- Only requires one memory module
- Only requires one path from CPU to memory for addresses
- Only requires one path from CPU to memory for data

The path is often a *bus*: a connection between two or more computer components. Today, it is commonly a set of electrical wires.

Program counter (PC)

- Keeps track of current machine instruction (i.e., PC)
- *Implicitly* points to next machine instruction (i.e., PC + 1)
- Can save and restore the PC to return a point of execution

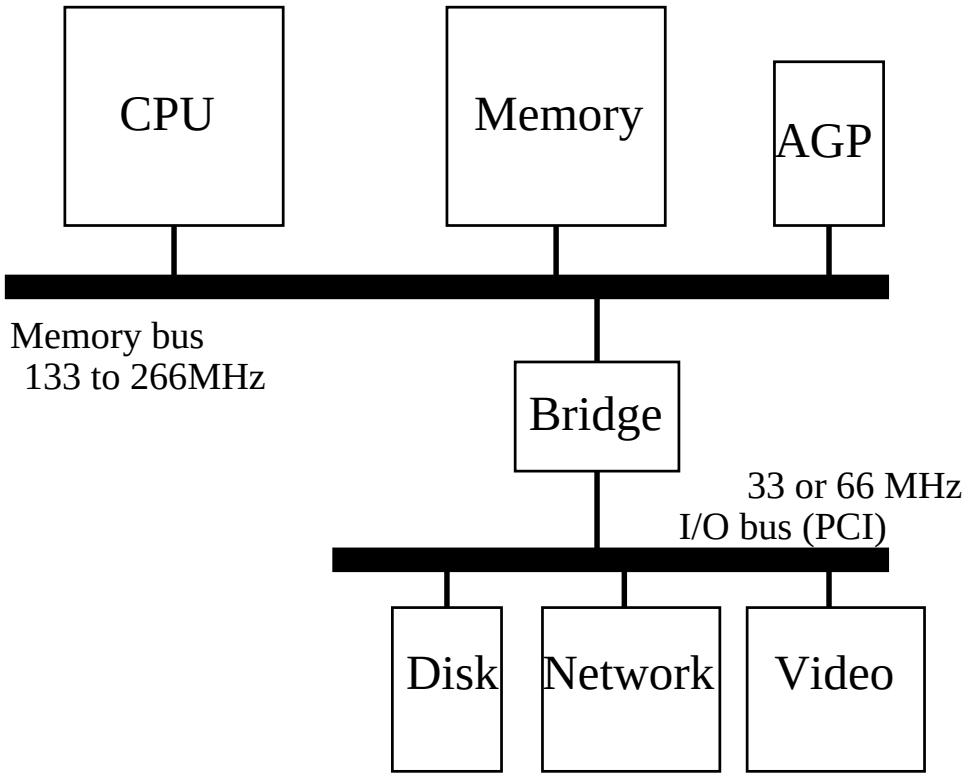
Alternatively, each machine instruction could *explicitly* indicate the next instruction to execute.

repeat

1. Fetch/get instruction at PC
2. Execute instruction
3. PC := PC + 1

forever

Typical Architecture



Number Representation

- Computers operate with `0`'s and `1`'s
- But, people prefer decimal numbers like 127, -14 (i.e., negative number)

How do we represent decimal numbers in binary form? (Negative/signed numbers come later.)

- First, let us deconstruct 127.

$$127 = (1 \times 10^2) + (2 \times 10^1) + (7 \times 10^0)$$

- More abstractly, 127 is a base 10 number and the value of the ith digit d is:

$$d \times base^i$$

Different Bases

- For a given *base*, the valid digits in the number system are between 0 and $base - 1$
- If $base = 2 \Longrightarrow$ binary number
  - Valid digits are { 0, 1 }.
  - For example, $1011_2 = 11$ decimal.
- If $base = 8 \Longrightarrow$ octal number
  - Valid digits are { 0, 1, 2, 3, 4, 5, 6, 7 }
  - For example, $37_8 = 31$ decimal.
- If $base = 10 \Longrightarrow$ decimal number
  - Valid digits are { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- If $base = 16 \Longrightarrow$ hexadecimal number
  - Valid digits are { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }
  - For example, $\mathrm{FFE}_{16} = 4094$ decimal.

Binary to Decimal Conversion

What is binary number 101_2 in decimal?

Answer:

$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

$= 4 + 0 + 1$

$= 5_{10}$

What is binary number 101101_2 in decimal?

Answer:

$101101_2 =$
$(1\times2^5)+(0\times2^4)+(1\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)$

$= 32 + 8 + 4 + 1$

$= 45_{10}$

Largest Numbers

What is the largest decimal number with 3 digits?

Answer:

Easy, 999. Or, $10^3 - 1$.

What is the largest binary number with 3 digits (expressed in decimal)?

Easy, 111_2. Or, $2^3 - 1$.

$111_2 = (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 4 + 2 + 1$

$= 7_{10}$

$= 2^3 - 1$

With n binary digits, the largest number is $2^n - 1$.

For n digits of base *base*, the largest number is $base^n - 1$.

32-bit Unsigned Numbers

On our MIPS CPU architecture, each register has 32 bits. What is the largest (unsigned) number?

Answer: $2^{32} - 1 = 4{,}294{,}967{,}295$.

NOTE:

$2^{32} = 2^2 \times 2^{30}$

$= 2^2 \times 2^{10} \times 2^{10} \times 2^{10}$, where $2^{10} = 1024$

$= 4(1024)^3 =$ approx. $4{,}000{,}000{,}000$

- HINT: Memorize various powers of 2 up to 2^{10}.

Overflow

What happens when we try to store 4,294,967,295 plus 1 in an unsigned 32-bit register?

Answer:

Overflow (and wrap around, like your car's odometer)

- Lesson: Computers have limits in the numbers that they can represent. Be aware of this, or risk getting wrong answers.

With a 64-bit register, what is the largest possible (unsigned) number?

Why are 64-bit computers becoming more common?

Decimal to Binary Conversion

What is 23_{10} in binary?

There are 2 common methods:

1. Express decimal as sum of distinct powers of 2.
   - $23 = 16 + 7 = 16 + 4 + 2 + 1$
   - $= (1 \times 2^4) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
   - Read off the coefficients: 10111_2
2. Repeated long division
   - $23/2 = 11$ remainder 1
   - $11/2 = 5$ remainder 1
   - $5/2 = 2$ remainder 1
   - $2/2 = 1$ remainder 0
   - $1/2 = 0$ remainder 1
   - Read off the remainders: 10111_2

Hexadecimal Numbers

- If $base = 16 \Longrightarrow$ hexadecimal number
  - Valid digits are { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }
  - For example, $\mathrm{FFE}_{16} = 4094$ decimal.

How do we convert FFE_{16} to decimal?

Answer:

$\mathrm{FFE}_{16} = (15 \times 16^2) + (15 \times 16^1) + (14 \times 16^0)$

$= (15 \times 256) + (15 \times 16) + (14 \times 1)$

$= 3840 + 240 + 14$

$= 4094_{10}$

What is 204_{16} in decimal?

Answer:

$204_{16} = (2 \times 16^2) + (0 \times 16^1) + (4 \times 16^0)$

$= (2 \times 256) + (0 \times 16) + (4 \times 1)$

$= 512 + 0 + 4$

$= 516_{10}$

If

```
A = 10  C = 12  E = 14
B = 11  D = 13  F = 15
```

What is DEADBEEF_{16} in decimal?

Answer:

DEADBEEF_{16}

$= (D : 13 \times 16^7) + (E : 14 \times 16^6) + (A : 10 \times 16^5) + (D : 13 \times 16^4) + (B : 11 \times 16^3) + (E : 14 \times 16^2) + (E : 14 \times 16^1) + (F : 15 \times 16^0)$

$= (D : 3,489,660,928) + (E : 234,881,024) + (A : 10,485,760) + (D : 851,968) + (B : 45,056) + (E : 3,584) + (E : 224) + (F : 15)$

$= 3,735,928,559_{10}$

- DEADBEEF_{16} is often used as an "initial value" for memory

Hex to Binary Conversion

This is a lot easier than hex to decimal, or binary to decimal.

Why?

Because the base for hex (base = 16) and binary (base = 2) are powers of 2. Therefore, each hexadecimal digit is exactly 4 bits.

Dec	Hex	Binary	Dec	Hex	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

1. Convert hex to binary by expressing number as distinct powers of 2

   $A2_{16} = 162_{10}$

   $= 128_{10} + 32_{10} + 2_{10}$

   $= 2^7 + 2^5 + 2^1$ (decimal numbers)

   $= 10100010_2$

   NOTE: A = 1010, 2 = 0010

2. Do a table look-up on each hexadecimal digit

   What is $DEADBEEF_{16}$ in binary?

   Answer:
   1101 1110 1010 1101 1011 1110 1110 1111

   What is $A7FB_{16}$ in binary?

   Answer:
   1010 0111 1111 1011

Therefore, a 32-bit register can hold 8 hexadecimal digits (e.g., $DEADBEEF_{16}$).

Addition and Subtraction in Binary

Example of (plain) addition:

```
Decimal     Binary

  5          0101
+ 6        + 0110
----       -------
 11          1011
```

Example of (plain) subtraction:

```
Decimal     Binary

  6          0110
- 5        - 0101
----       -------
  1          0001
```

NOTE: A "borrow" equals 2.

Negative Numbers

So far, we have ignored numbers like -14, -1, and -223,433.

How can we encode these numbers in binary format (or hexadecimal)?

1. Signed magnitude representation

   Basically, we use one bit for the sign.
   0 = positive, 1 = negative.

   So, -14 = 1 000 1110

   If 1 bit is used for the sign, then we have 7 bits left over for the number itself (aka magnitude).

   So, the smallest number we can represent with 8 bits using signed magnitude is $2^7 - 1$ $\Longrightarrow$ -127

2. Two's complement representation

2's Complement

Given a number N to be encoded in n bits, its 2's complement ($\tilde{N}$) is defined as:

$$\tilde{N} = 2^n - N$$

Example: Assume $n = 4 \Longrightarrow 2^n = 16$,

1. If $N = 3 = 0011_2$,
   $\tilde{3} = 16 - 3 = 13$
   $= 1101_2$
2. If $N = 6 = 0110_2$,
   $\tilde{6} = 16 - 6 = 10$
   $= 1010_2$
3. If $N = 2 = 0010_2$,
   $\tilde{2} = 16 - 2 = 14$
   $= 1110_2$

Notice the following properties:

1. $\tilde{\tilde{N}} = N$

   Why? $\tilde{\tilde{N}} = 2^n - (\tilde{N})$

   $= 2^n - (2^n - N)$

   $= N$

2. $\tilde{N} = \bar{N} + 1$ (NOTE: Tilde-N and Bar-N)

   where $\bar{N}$ is called the 1's complement, or bitwise negation

   For example, if $N = 2 = 0010_2$,

   $\bar{2} = 00\bar{1}0_2 = 1101_2$

   Therefore, $\tilde{2} = \bar{2} + 1$

   $= 1101_2 + 0001_2$

   $= 1110_2$

3. $\tilde{N} + N = 0$, with carry out

   For example, if $N = 2 = 0010_2$,

   $\tilde{2} + 2 = 1110_2 + 0010_2$

   $= 0$, with carry out

+/- with 2's Complement

In practice, we do the following:

1. If $N \geq 0$, use "plain" binary representation.

   For example, $+2 = 0010_2$

2. If $N < 0$, use 2's complement of the absolute value

   For example, $-2 = 1110_2$

Recall that $2 - 2 = 2 + (-2) =$

```
  Decimal    Binary

    2          0010
+  -2      +   1110
  ----      -------
    0          0000
```

$\Longrightarrow$ Subtraction is the same as taking the 2's complement of the second number and then adding!

Also, adding numbers, whether positive or negative, works as expected.

Advantages of 2's complement

1. Subtraction can be done in terms of addition.

   $A - B = A + (-B) = A + \tilde{B}$

   NOTE: This works even if A or B is already negative and in 2's complement form.

2. We do not need to check the sign bit (as with signed magnitude) and then compare magnitudes.

Example of subtraction, $3 - 2$,
$(\tilde{2} = 00\bar{1}0_2 + 1_2 = 1101_2 + 1_2 = 1110_2)$

Decimal	Binary
3	0011
- 2	+ 1110
----	-------
1	0001

Example of subtraction, $14 - 5$.

1. $14 = 0000\ 1110_2$
2. $-5 = \tilde{5}$

   Easiest to logically negate and add 1.

   $= 0000\tilde{}0101$

   $= 1111\ 1010_2 + 1_2$

   $= 1111\ 1011_2$

Decimal	Binary
14	0000 1110
- 5	+ 1111 1011
----	----------
9	0000 1001

Common Patterns

Since $n = 32$ on our MIPS architecture,

$-1 = \tilde{1}$

$= \texttt{0000 0000 0000 0000}\tilde{}\texttt{0000 0000 0000 0001}_2$

$= \texttt{1111 1111 1111 1111 1111 1111 1111 1111}_2$

$=$ FFFF FFFF

$-2 = \texttt{1111 1111 1111 1111 1111 1111 1111 1110}_2$

$-3 = \texttt{1111 1111 1111 1111 1111 1111 1111 1101}_2$

Largest number is $2,147,483,647 =$
$\texttt{0111 1111 1111 1111 1111 1111 1111 1111}_2$

Smallest number is $-2,147,483,648 =$
$\texttt{1000 0000 0000 0000 0000 0000 0000 0000}_2$

- All positive numbers have `0` as the most significant bit.
- All negative numbers have `1` as the most significant bit.

Question: If 1010_2 is a number in 2's complement, what is its value in decimal?

1. Since the most significant bit is 1, we know it is a negative number.
2. To find out the magnitude, we simply take the 2's complement (recall that: $\tilde{\tilde{N}} = N$)

   $\tilde{1010}_2$

   $= \bar{1010}_2 + 1_2$

   $= 0101_2 + 1_2$

   $= 0110_2$

   $= 6_{10}$

$\Longrightarrow$ -6

Range of Numbers

So, if $n = 32$ bits, then the 2's complement representation allows a 32-bit binary number to hold the integer numbers N in the range:

$$-2,147,483,648 \leq N \leq 2,147,483,647$$

In general,

$$-2^{n-1} \leq N \leq 2^{n-1} - 1$$

NOTE:

1. There is one "extra" negative number because the 0 (zero) takes up a value.
2. Overflow is possible, as with unsigned numbers.

```
Decimal    Binary
  5          0101
+ 6        + 0110
----       -------
 11          1011
```

No overflow for unsigned. Yes, overflow for 2's complement. Why?

Logical Operations

- Chapter 4.4 and Cmput 272

In addition to arithmetic, computers are good at logical operations on binary values: logical-AND, logical-OR, logical-NOT (or just AND, OR, NOT for short).

The output of a logical operation is a result of the input according to the following *truth table* for values `A` and `B`.

		AND	OR	NOT
A	B	A & B	A v B	~A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

0 = false, 1 = true

We have already seen bitwise logical-negation with 1's complement.

Other examples:

```
                AND       OR       NOT
 A   B       A & B     A v B      ~A
-------------------------------------
 0   0         0         0         1
 0   1         0         1         1
 1   0         0         1         0
 1   1         1         1         0
```

```
    0101        0101        0101
  & 0110      v 0110      ~
  -------     -------     ------
    0100        0111        1010
```

What are logical operations used for?

They are used for manipulating individual bits within a word (i.e., 32-bit value).

1. AND is often used to clear bits using a *mask*.

```
  0101      0101
& 1100    & 0011
-------   -------
  0100      0001
```

2. OR is often used to set bits.

```
  0101      0101
v 0010    v 1000
-------   -------
  0111      1101
```

Using hexadecimal numbers, suppose `A` = `0x000F` (`0x` is a common prefix for hexadecimal) and `B` = `0xABCD`.

Then

```
  0x000F        0x000F
& 0xABCD      v 0xABCD
---------     ---------
  0x000D        0xABCF
```

Masks can be used to dissect and reconstruct numbers:

```
  0x00FF        0xFF00        0xAB00
& 0xABCD      & 0xABCD      v 0x00CD
---------     ---------     ---------
  0x00CD        0xAB00        0xABCD
```

Suppose I want to set the hexadecimal digit at position 1 to be `E` for any number.

How can I do this using logical operations?

If the original number is `0x1234`, then:

```
Step 1    0x1234    Step 2    0x1204
        & 0xFF0F            v 0x00E0
        ---------           ---------
          0x1204              0x12E4
```

Why does using OR in a single step not work?

If the original number is `0xABCD`, then:

```
Step 1    0xABCD    Step 2    0xAB0D
        & 0xFF0F            v 0x00E0
        ---------           ---------
          0xAB0D              0xABED
```

Shift Operations

Usually, there are assembly language instructions for moving all bits of a word one radix to the left or the right.

1. Logical shift right (in C "`>>`").

```
>> 0110      >> 0011
-------     -------
  0011        0001
```

   Shifting right is equivalent to dividing a number by 2.

2. Logical shift left (in C "`<<`").

```
<< 0110      << 0011
-------     -------
  1100        0110
```

   Shifting left is equivalent to multiplying a number by 2.

NOTE: Be careful of signed numbers when shifting (Chapter 4.10)!

ASCII Characters

Keyboard input is read as ASCII characters, each requiring one byte of storage.

The key `A` on the keyboard generates a specific number, namely decimal 65. Lowercase `a` is number 97.

Compare this to type `char` in C.

From `man ascii` under Unix. The numbers are decimal values.

32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

Arrays of characters, or strings, make up the vast majority of input and output intended for human consumption.

Strings are null terminated (i.e., the last byte of data is a zero).

MIPS Assembly Language

Key concepts:

1. There are 32 general-purpose registers, each 32-bits wide (`r0` to `r31`)
2. The *same* 32 general-purpose registers have aliases:
   (a) `v0`, `v1`: result of expression or function
   (b) `a0` to `a3`: arguments
   (c) `t0` to `t9`: temporary
   (d) `s0` to `s7`: saved temporary
   (e) `$zero`: always set to zero
   (f) others: `at`, `Hi`, `Lo`, `sp`,
3. There are 32 floating-point registers (`f0` to `f31`). We will not discuss any further for now.
4. All arithmetic and logical computation occurs between registers. Data must first be moved into a register.

5. The MIPS assembler (and SPIM/XSPIM) provides extra functionality not directly provided by the hardware.
   (a) Data layout directives
   (b) Pseudoinstructions (Careful!)
6. The SPIM/XSPIM environment provides extra functionality
   (a) Input/output functions
   (b) XSPIM provides debugging capabilities

Example: Hello, world

```
# How to run:  spim -file ex1.s, or use xspim
        .data
str1:
        .asciiz "Hello, world\n"

        .text
        .globl main
main:
        li      $v0, 4          # print_string
        la      $a0, str1
        syscall
```

1. Comments start with `#` and go to end-of-line
2. Data (`.data`) and code segments (`.text`)
3. Data layout (`.asciiz`)
4. Global labels (`.globl`)
5. Assembly language instructions and pseudoinstructions
6. Function call to SPIM/XSPIM (`syscall`)

The pseudoinstructions are:

1. `li $v0, 4`: load immediate value

   `v0` = 4 (load `v0` with constant 4)

2. `la $a0, str1`: load address

   `a0` = address of `str1` (load `a0` with address)

Load the file into XSPIM. It is clear that these are pseudoinstructions.

```
From XSPIM's Text Segment Window:

[0x00400020] 0x34020004  ori $2, $0, 4           ; 10: li $v0, 4 # print_string
[0x00400024] 0x3c041001  lui $4, 4097 [str1]     ; 11: la $a0, str1
[0x00400028] 0x0000000c  syscall                 ; 12: syscall
```

If we single step to after line 11, we see:

```
From XSPIM's Register/Top Window:

R4 (a0) = 10010000
```

NOTE: $4097_{10} = 1001_{16}$

```
From XSPIM's Data Segments Window:

[0x10010000]                    0x6c6c6548  0x77202c6f  0x646c726f  0x0000000a
[0x10010010]...[0x10020000]     0x00000000
```

...which is ASCII.

Basics of Assembly Language

There are 4 main groups of *instructions* (specified by *mnemonics*) for any CPU architecture:

1. **Data transfer** (aka load, store)

   e.g. `lw`, `sw`, `la`, `lb`, `lbu`, `sb`, `move`

2. **Computation** (between registers)
   (a) Arithmetic, e.g. `add`, `sub`, `addi`, `addu`
   (b) Logical, e.g. `and`, `or`, `sll`, `srl`

3. **Control flow change** (affects PC)
   (a) Conditional branch, e.g. `beq`, `bne`
   (b) Unconditional branch, e.g. `b`, `j`

4. **Privileged instructions** (for the OS)

NOTE: Mnemonics are encoded into binary *opcodes* by the assembler.

Basics of Addressing Modes

Instructions operate on *operands*, or "arguments".

There are 4 general categories (compare with pages 151, A-50):

1. **Immediate**: use a constant data value
2. **Absolute**: use data from memory
   - "address of label" on MIPS
   - "address of label + or - immediate" on MIPS
3. **Register direct**: use data *in* a register
   - "contents of register" on MIPS, register addressing
   - PC-relative addressing, Pseudodirect addressing
4. **Register indirect**: use data *pointed to* by a register
   - "immediate + contents of register" on MIPS
   - "address of label + or - (immediate + contents of register)" on MIPS
   - base or displacement addressing

Be careful: Absolute addressing on the MIPS architecture is implemented by the assembler and **NOT** by the hardware.

```
        .text
        lb   $s0, strsingle      # Load byte into $s0
                                 # NOT la $s0,strsingle


        .data
strsingle:
        .asciiz "  "
```

```
From XSPIM's Text Segments Window:

[0x00400034] 0x3c011001  lui $1, 4097          ; 70: lb    $s0, strsingle
[0x00400038] 0x80300000  lb $16, 0($1)
```

Things to note:

1. `$1` is the same as register `$at`, which is reserved for the assembler's use
2. In fact, the only memory-addressing mode implemented by the CPU hardware is register indirect: `C(rx)`

Example: Count

```
# How to run:  spim -file ex2.count.s, or use xspim
# Program by John Waldron

        .text
        .globl main
main:
        li $t1,0        # $t1 will be the array index
        li $t2,0        # $t2 will be the counter
        lb $t3,char     # and $t3 will hold the char

loop:   lb $t0,str($t1) # fetch next char
        beqz $t0,strEnd # if it's a null, exit loop
        bne $t0,$t3,con # not null; same as char?
        add $t2,$t2,1   # yes,increment counter
con:    add $t1,$t1,1   # increase index
        j loop          # and continue

strEnd:
        la $a0,ans      # system call to print
        li $v0,4        # out a message
        syscall

        move $a0,$t2    # system call to print
        li $v0,1        # out the count worked out
        syscall
```

continued...

```
        la $a0,endl     # system call to print
        li $v0,4        # out a newline
        syscall

        li $v0,10
        syscall         # au revoir...

        .data
str:    .asciiz "abceebceebeebbacacb"
char:   .asciiz "e"
ans:    .asciiz "Count is "
endl:   .asciiz "\n"
```

```
unix-prompt% spim -file ex2.count.s
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ./trap.handler
Count is 6
```

Example: Dec to Hex

```
# How to run:  spim -file ex3.dec2hex.s, or use xspim
# Program by John Waldron
# Purpose:  ask user for decimal number,
#  convert to hex, print the result.
        .text
        .globl main
main:
        la $a0,prompt   # print prompt on terminal
        li $v0,4
        syscall

        li $v0,5        # syscall 5 reads an integer
        syscall
        move $t2,$v0    # $t2 holds hex number

        la $a0,ans1     # print string before result
        li $v0,4
        syscall
```

continued...

```
##      t0 - count for 8 digits in word
##      t1 - each hex digit in turn
##      t2 - number read in
##      t3 - address of area used to set up answer string

        li $t0,8        # eight hex digits in word
        la $t3,result   # answer string set up here

loop:   rol $t2,$t2,4   # start with leftmost digit
        and $t1,$t2,0xf # mask one digit
        ble $t1,9,print # check if 0 to 9
        add $t1,$t1,7   # 7 chars between '9' and 'A'
print:  add $t1,$t1,48  # ASCII '0' is 48
        sb $t1,($t3)    # save in string
        add $t3,$t3,1   # advance destination pointer
        add $t0,$t0,-1  # decrement counter
        bnez $t0,loop   # and continue if counter>0

        la $a0,result   # print result on terminal
        li $v0,4
        syscall

        li $v0,10
        syscall         # au revoir...

        .data
result: .space 8
        .asciiz "\n"
prompt: .asciiz "Enter decimal number: "
ans1:   .asciiz "Hexadecimal is "
```

```
unix-prompt% spim -file ex3.dec2hex.s
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ./trap.handler
Enter decimal number: 16
Hexadecimal is 00000010

...

Enter decimal number: 334
Hexadecimal is 0000014E

...

Enter decimal number: 299
Hexadecimal is 0000012B
```

Still have to be careful of pseudoinstructions.

It is also very interesting to see how pseudoinstructions are translated into real instructions.

```
From XSPIM's Text Segments Window:

[0x00400058] 0x000a0f02  srl $1, $10, 28                ; 28: rol $t2,$t2,4
[0x0040005c] 0x000a5100  sll $10, $10, 4
[0x00400060] 0x01415025  or $10, $10, $1
[0x00400064] 0x3149000f  andi $9, $10, 15               ; 29: and $t1,$t2,0xf
[0x00400068] 0x2921000a  slti $1, $9, 10                ; 30: ble $t1,9,print
[0x0040006c] 0x14200002  bne $1, $0, 8 [print-0x0040006c]
[0x00400070] 0x21290007  addi $9, $9, 7                 ; 31: add $t1,$t1,7
[0x00400074] 0x21290030  addi $9, $9, 48                ; 32: add $t1,$t1,48
```

- `$1` is the same as register `$at`, which is reserved for the assembler's use

Endianness

The order in which we store the individual bytes of a multi-byte value is somewhat arbitrary. Consider `0x76543210` (a 32-bit, 4-byte value) stored at address `0x10010000`.

1. **big endian**: most significant byte (MSB) to least significant byte (LSB)

`0x10010000`	`0x76`
`0x10010001`	`0x54`
`0x10010002`	`0x32`
`0x10010003`	`0x10`

2. **little endian**: LSB to MSB

`0x10010000`	`0x10`
`0x10010001`	`0x32`
`0x10010002`	`0x54`
`0x10010003`	`0x76`

- The MIPS, SPARC, Motorola 68xxx CPUs are big endian.
- The Intel x86 family is little endian.

Consider this simple program:

```
# How to run:  spim -file ex4.endian.s, or use xspim

        .text
        .globl main
main:
        li $t0,0        # clear registers
        li $t1,0
        li $t2,0
        li $t3,0

        lbu $t0,value1  # load byte-by-byte
        lbu $t1,value1+1
        lbu $t2,value1+2
        lbu $t3,value1+3

        li $v0,10
        syscall         # au revoir...

        .data
value1:
        .word   0x76543210
```

```
From XSPIM's Data Segments Window:

        DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x76543210  0x00000000  0x00000000  0x00000000
```

```
From XSPIM's Data Segments Window:

        DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x76543210  0x00000000  0x00000000  0x00000000
```

We look at the contents of registers after all the `lbus`.

On an Intel/x86 box (`csu401.cs`):

```
From XSPIM's Register/Top Window:

                                General Registers
R0 (r0) = 00000000  R8  (t0) = 00000010  R16 (s0) = 00000000  R24 (t8) = 00000000
R1 (at) = 10010000  R9  (t1) = 00000032  R17 (s1) = 00000000  R25 (t9) = 00000000
R2 (v0) = 00000004  R10 (t2) = 00000054  R18 (s2) = 00000000  R26 (k0) = 00000000
R3 (v1) = 00000000  R11 (t3) = 00000076  R19 (s3) = 00000000  R27 (k1) = 00000000
```

On a Solaris 8/SPARC box (`csu501.cs`):

```
From XSPIM's Register/Top Window:

                                General Registers
R0 (r0) = 00000000  R8  (t0) = 00000076  R16 (s0) = 00000000  R24 (t8) = 00000000
R1 (at) = 10010000  R9  (t1) = 00000054  R17 (s1) = 00000000  R25 (t9) = 00000000
R2 (v0) = 00000004  R10 (t2) = 00000032  R18 (s2) = 00000000  R26 (k0) = 00000000
R3 (v1) = 00000000  R11 (t3) = 00000010  R19 (s3) = 00000000  R27 (k1) = 00000000
```

Conclusion? Be careful of endianness and which machine you use for your assignments!

Again, consider `0x76543210` (a 32-bit, 4-byte value) stored at address `0x10010000`.

Why does memory **not** look like this (little-endian)?

`0x10010000`	`0x01`
`0x10010001`	`0x23`
`0x10010002`	`0x45`
`0x10010003`	`0x67`

Arrays and array indexing

There is no type system (e.g., C's `int`, `char`) in assembly language.

The machine is byte addressable, not integer addressable.

The assembler provides data directives to help layout data in the data segment, but the programmer is responsible for manipulating pointers and addresses according to the size of the data item in an array.

For example, the assembler provides `.word` and `.byte` data directives.

NOTE: On our systems,

```
sizeof( int ) == 4
```

```
sizeof( char ) == 1
```

```
# How to run:  spim -file ex5.intarray.s, or use xspim
        .text
        .globl main
main:
        li $t0,0        # clear sum
        la $t1,intarray # array of ints in memory

loop:
        lw $t2, ($t1)   # for each int
        beq $t2,0xff,printsum
        add $t0,$t0,$t2 # accumulate
        add $t1,$t1,4   # **** next int
        b loop

printsum:
        li $v0,1        # print_int
        move $a0,$t0
        syscall
        li $v0,4        # print_string
        la $a0,newline
        syscall

        li $v0,10
        syscall         # au revoir...

        .data
intarray:
        .word   0x01, 0x02, 0x03, 0x04, 0x05
        .word   0xff    # array terminator
newline:
        .asciiz "\n"
```

For the array of integers:

```
From XSPIM's Data Segments Window:

        DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x00000001  0x00000002  0x00000003  0x00000004
[0x10010010]                    0x00000005  0x000000ff  0x0000000a  0x00000000
```

```
unix-prompt% spim -file ex5.intarray.s
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ./trap.handler
15
```

```
# How to run:  spim -file ex6.bytearray.s, or use xspim
        .text
        .globl main
main:
        li $t0,0        # clear sum
        la $t1,bytearray  # array of bytes in memory

loop:
        lbu $t2, ($t1)  # for each byte
        beq $t2,0xff,printsum
        add $t0,$t0,$t2 # accumulate
        add $t1,$t1,1   # **** next byte
        b loop

printsum:
        li $v0,1        # print_int
        move $a0,$t0
        syscall
        li $v0,4        # print_string
        la $a0,newline
        syscall

        li $v0,10
        syscall         # au revoir...

        .data
bytearray:
        .byte   0x01, 0x02, 0x03, 0x04, 0x05
        .byte   0xff    # array terminator
newline:
        .asciiz "\n"
```

For the array of bytes on an Intel x86 box:

```
From XSPIM's Data Segments Window:

        DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x04030201  0x000aff05  0x00000000  0x00000000
[0x10010010]...[0x10020000]     0x00000000
```

NOTE: The little endianness of x86 makes the data look "funny"!

```
unix-prompt% spim -file ex5.intarray.s
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ./trap.handler
15
```

Array Indexing: Days in a month

```
# How to run:  spim -file ex7.daysinmonth.s, or use xspim

        .text
        .globl main
main:
        li $v0,4        # print_string
        la $a0,promptmonth
        syscall

        li $v0,5        # read_int
        syscall

        move $s0,$v0    # save month (in $sX)

        li $v0,4        # print_string
        la $a0,outphrase
        syscall
```

```
        sub $s0,$s0,1   # arrays start at index 0
        sll $t1,$s0,2   # multiply by 4 wrt integer
        lw $t2, month2days($t1)         # table lookup

        li $v0,1        # print_int
        move $a0,$t2
        syscall

        li $v0,4        # print_string
        la $a0,endline
        syscall

        li $v0,10
        syscall         # au revoir...

        .data
month2days:
#               Jan  Feb  Mar  Apr  May  June
        .word   31,  29,  31,  30,  31,  30

#               July Aug  Sept Oct  Nov  Dec
        .word   31,  31,  30,  31,  30,  31
promptmonth:
        .asciiz "What month in year 2000 (1 to 12)? "
outphrase:
        .asciiz "In that month, there are "
endline:
        .asciiz " days.\n"
```

```
unix-prompt% spim -file ex7.daysinmonth.s
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ./trap.handler
What month in year 2000 (1 to 12)? 1
In that month, there are 31 days.

...

What month in year 2000 (1 to 12)? 2
In that month, there are 29 days.

...

What month in year 2000 (1 to 12)? 3
In that month, there are 31 days.

...

What month in year 2000 (1 to 12)? 6
In that month, there are 30 days.

...

What month in year 2000 (1 to 12)? 12
In that month, there are 31 days.
```

Simple Subroutines: jal, jr $ra

```
# How to run:  spim -file ex8.simple.sub.s, or use xspim
# Based on the days-in-month example

        .text
        .globl main
main:
        jal getinput    # simple subroutine call
        move $s0,$v0    # save month (in $sX)

        sub $s0,$s0,1   # arrays start at index 0
        sll $t1,$s0,2   # multiply by 4 wrt integer
        lw $a0, month2days($t1)         # table lookup

        jal outputdays

        li $v0,10
        syscall         # au revoir...
```

```
getinput: #-------------------------------------------
        li $v0,4        # print_string
        la $a0,promptmonth
        syscall

        li $v0,5        # read_int
        syscall

        jr $ra          # return to caller

outputdays: #-----------------------------------------
        move $s0,$a0    # Save number of days

        li $v0,4        # print_string
        la $a0,outphrase
        syscall

        move $a0,$s0    # Restore number of days
        li $v0,1        # print_int
        syscall

        li $v0,4        # print_string
        la $a0,endline  # simple newline
        syscall

        jr $ra          # return to caller
```

```
        .data
month2days:
#               Jan  Feb  Mar  Apr  May  June
        .word   31,  29,  31,  30,  31,  30
#               July Aug  Sept Oct  Nov  Dec
        .word   31,  31,  30,  31,  30,  31
promptmonth:
        .asciiz "What month in year 2000 (1 to 12)? "
outphrase:
        .asciiz "In that month, there are "
endline:
        .asciiz " days.\n"
```

Subroutines and the Stack

Consider the C code:

```
int fact( int n )
{
        if( n < 1 )
                return( 1 );
        else
                return( n * fact( n - 1 ) );
}
```

NOTE:

1. It is recursive.

   How do we keep track of the different return addresses for `jr $ra`?

2. It has a parameter `n`.

   How do we pass an arbitrary number of parameters?

3. `n` is also a local variable

   How do we allocate storage for local variables, one set for each recursive call?

Answer: Use the stack.

```
# How to run:  spim -file ex9.factorial.s, or use xspim
# Based on textbook page A-26 to A-29

        .text
        .globl main
main:
    # Handle callee-saved registers (except $sX, oops)
        subu $sp,$sp,32 # Push stack frame, grows down
        sw $ra,20($sp)  # ...return address (20--23)
        sw $fp,16($sp)  # ...frame pointer (16--19)
        addu $fp,$sp,28 # New frame pointer (28--31)

        jal getinput    # Get input from user, n
        move $s0,$v0    # Save it
        move $a0,$v0    # Use it as parameter
        jal fact        # Compute n!

        move $a0,$s0    # Value n
        move $a1,$v0    # Value n!
        jal printans    # Print them

    # Restore registers and stack frame
        lw $ra,20($sp)  # ...return address
        lw $fp,16($sp)  # ...frame pointer
        addu $sp,$sp,32 # Pop stack frame
        jr $ra          # Return to caller (trap.handler)
```

```
fact:
    # Handle callee-saved registers
        subu $sp,$sp,32 # Push stack frame, grows down
        sw $ra,20($sp)  # ...return address (20--23)
        sw $fp,16($sp)  # ...frame pointer (16--19)
        addu $fp,$sp,28 # New frame pointer (28--31)

        sw $a0,0($fp)   # Save argument (n) (28--31)

    # Inside the function...
        lw $v0,0($fp)   # Load n
        bgtz $v0,$L2    # Branch if n > 0
        li $v0,1        # Return value 1
        j $L1           # Clean-up stack

$L2:
        lw $v1,0($fp)   # Load n (inefficient?)
        subu $v0,$v1,1  # Compute (n-1)
        move $a0,$v0    # Argument for recursive call
        jal fact        # Recurse.
                        # Return value in $v0
        lw $v1,0($fp)   # Re-load n
        mul $v0,$v0,$v1 # Compute fact(n-1) * n

$L1:
    # Restore registers and stack frame
        lw $ra,20($sp)  # ...return address
        lw $fp,16($sp)  # ...frame pointer
        addu $sp,$sp,32 # Pop stack frame
                        # Result is in $v0
        jr $ra          # Return to caller
```

```
getinput:
        li $v0,4        # print_string
        la $a0,promptnum
        syscall

        li $v0,5        # read_int
        syscall
                        # Return value in $v0
        jr $ra          # return to caller

# NOTE:  Data segment between text segments
# Be sure to start next text segment with .text
        .data
promptnum:
        .asciiz "Compute n! for what n? "
```

```
        .text
printans:
    # A simple subroutine, should save $sX but don't
        move $s0,$a0    # Save n for now
        move $s1,$a1    # Save n! for now
        move $s2,$ra    # Save n! for now

        la $a0, preans  # Print answer header
        li $v0,4        # print_string
        syscall
        move $a0,$s0    # Restore n
        li $v0,1        # print_int
        syscall
        la $a0, postans # Print more of answer
        li $v0,4        # print_string
        syscall
        move $a0,$s1    # Restore n!
        li $v0,1        # print_int
        syscall
        la $a0, newline
        li $v0,4        # print_string
        syscall

        move $ra,$s2    # Restore return address
        jr $ra

        .data
preans:  .asciiz "The factorial of "
postans: .asciiz " is "
newline: .asciiz "\n"
```

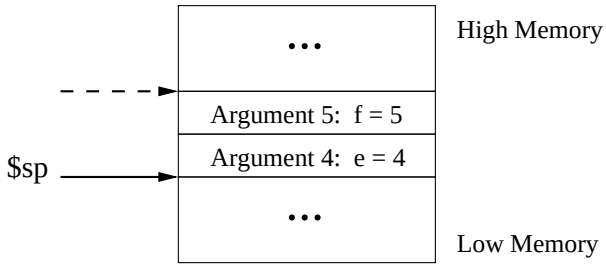
Parameters and local variables

```
int func( int a, int b, int c, int d, int e, int f )
{       /* Callee */
        int i, j;
        return 0;
}

main()
{       /* Caller */
        func( 0, 1, 2, 3, 4, 5 );  /* Call-site */
}
```

```
main:   subu $sp,$sp,4  # Push argument ( int f )
        li $t0,5        # ..by value, 5
        sw $t0,0($sp)

        subu $sp,$sp,4  # Push argument ( int e )
        li $t0,4        # ..by value, 4
        sw $t0,0($sp)

        li $a3, 3       # Pass by value in reg:  d = 3
        li $a2, 2       # Pass by value in reg:  c = 2
        li $a1, 1       # Pass by value in reg:  b = 1
        li $a0, 0       # Pass by value in reg:  a = 0

        jal func

        addu $sp,$sp,4  # Pop argument ( int e )
        addu $sp,$sp,4  # Pop argument ( int f )
```

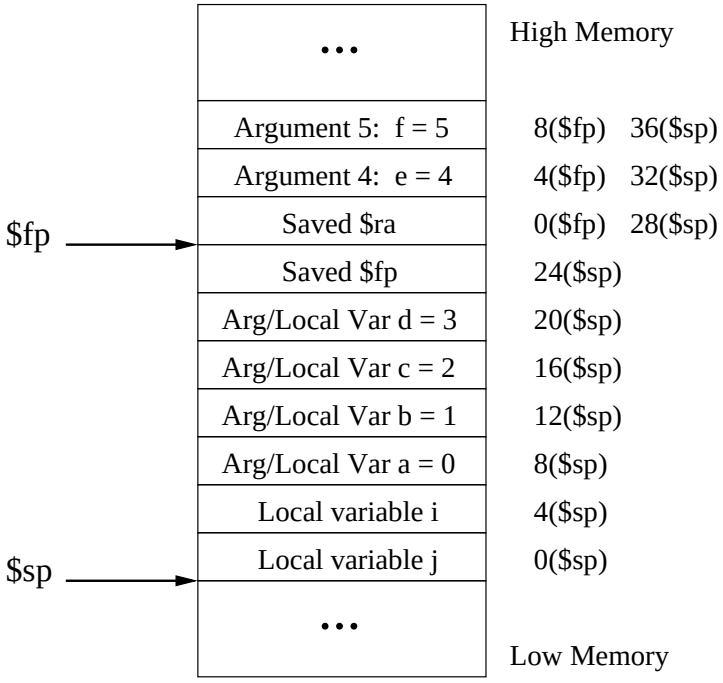
It can be simplified:

```
main:   subu $sp,$sp,8  # Grow stack for 2 parameters
        li $t0,5        # Push argument by value: f = 5
        sw $t0,4($sp)   #     NOTE:  the offset = 4
        li $t0,4        # Push argument by value: e = 4
        sw $t0,0($sp)

        li $a3, 3       # Pass by value in reg:  d = 3
        li $a2, 2       # Pass by value in reg:  c = 2
        li $a1, 1       # Pass by value in reg:  b = 1
        li $a0, 0       # Pass by value in reg:  a = 0

        jal func
                        # Caller cleans up arguments
        addu $sp,$sp,8  # Pop arguments e and f
```



NOTE: The layout of the stack frame here (and next slide) is different than the previous example, but consistent with Figure A-11, pg. A-25.

```
func:
    # Entry code

    # Handle callee-saved registers
        subu $sp,$sp,32 # Grow stack for 8 words
        sw $ra,28($sp)  # Save return address (28--31)
        sw $fp,24($sp)  # Save old frame pointer (24--27)
        addu $fp,$sp,28 # New frame pointer

    # Save parameters to stack (can be optimized out
    #   by always keeping variables in registers
    #   and never calling other subroutine, $aX not saved)
        sw $a0,8($sp)   # Save argument a
        sw $a1,12($sp)  # Save argument b
        sw $a2,16($sp)  # Save argument c
        sw $a3,20($sp)  # Save argument d
```

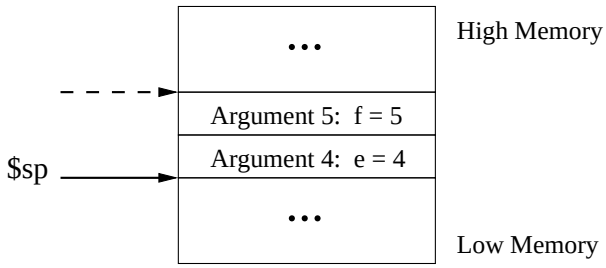


Note: Inside the code for `func`, the parameters and local variables are all at addresses that are relative to the stack pointer (i.e., `$sp`-relative addressing)

1. `int i` is `4($sp)`
2. `int j` is `0($sp)`
3. `int a` is `8($sp)`
4. `int b` is `12($sp)`
5. `int c` is `16($sp)`
6. `int d` is `20($sp)`
7. `int e` is `32($sp)` or `4($fp)`
8. `int f` is `36($sp)` or `8($fp)`

The exit code is a lot simpler...

```
func:

...snip...

    # Exit code

    # Restore registers and stack frame
        lw $ra,28($sp)  # Restore return address (28--31)
        lw $fp,24($sp)  # Restore frame pointer (24--27)
        addu $sp,$sp,32 # Shrink stack for 8 words

    # Assume return value is in $v0
        jr $ra
```



Look familiar?

Concluding Remarks on Subroutines

- Although subroutine calls have overheads, it is generally better to modularize your code
- Keep parameters and values in registers as much as possible
- Avoid *spilling* and *filling* registers to the stack, but it can be unavoidable since only finite number of registers
- A solid understanding of stack frames and activation records is important for Cmput 415 (Compilers), Cmput 425 (O-O Languages), Java VM
- Compilers for high-level languages, such as C, C++, Java, can optimize the above better than humans
  - inlining code
  - register allocation

C-style control flow

In various examples, we have already seen how to implement if-then control flow and loops in assembly.

A few quick examples to complete the picture:

1. if-then-else
2. for loop
3. while loop
4. switch and case

if-then-else

From pg. 124, textbook:

```
if( i == j )
        f = g + h;
else    f = g - h;
```

- Assume `f` is in `$s0`.
- Assume `g` is in `$s1`. Assume `h` is in `$s2`
- Assume `i` is in `$s3`. Assume `j` is in `$s4`

```
    # Test if condition:  ?? i == j ??
        bne $s3,$s4,Else        # if false, goto Else

    # Do "if" part
        add $s0,$s1,$s2         # f = g + h
        j Exit                  # skip over "else" part

Else:
    # Do "else" part
        sub $s0,$s1,$s2         # f = g - h

Exit:
```

for loop

```
/* for( init-top; test; bottom ) */
for( i = 0; i < j; i++ )
{
        /* Loop body */
}
```

- Assume `i` is in `$s0`. Assume `j` is in `$s1`.

```
Init-Top:
    # Initialize loop at top
        li $s0, 0               # i = 0

Test:
    # Test exit condition for loop:  ! ( i < j )
        bge $s0,$s1,Exit-For    # !!! Dual of < is >= !!!

    # Loop body

Bottom:
    # Bottom:  set up next iteration
        addi $s0,$s0,1          # i++
        j Test

Exit-For:
```

`while` loop

```
while( workToDo )
{
        /* Loop body */
}
```

- Assume `workToDo` is in `$s0`.

```
While-Test:
    # Test exit condition for loop
        beq $s0,$zero,Exit-While   # Test "opposite"/dual

    # Loop body

    # Unconditionally go back to top/while-test
        j While-Test
Exit-While:
```

switch-case with jump table

From pg. 129 of the textbook.

```
switch( k )
{
        case 0:  f = i + j; break; /* k = 0 */
        case 1:  f = g + h; break; /* k = 1 */
        case 2:  f = g - h; break; /* k = 2 */
        case 3:  f = i - j; break; /* k = 3 */
}
```

- Assume `f` is in `$s0`.
- Assume `g` is in `$s1`. Assume `h` is in `$s2`.
- Assume `i` is in `$s3`. Assume `j` is in `$s4`.
- Assume `k` is in `$s5`. Assume `$t2` holds 4.

```
switch( k )
{
        case 0:  f = i + j; break; /* k = 0 */
        case 1:  f = g + h; break; /* k = 1 */
        case 2:  f = g - h; break; /* k = 2 */
        case 3:  f = i - j; break; /* k = 3 */
}
```

```
    # Check the boundaries of values for k
        slt $t3,$s5,$zero       # Is k < 0 ?
        bne $t3,$zero,Exit-Switch   # bne => true, leave
        slt $t3,$s5,$t2         # Is k < 4 ($t2 = 4) ?
        beq $t3,$zero,Exit-Switch   # beq => false, leave

        sll $t7,$s5,2           # k*4

        add $t1,$t1,$t7         # $t1 = addr JmpTable[k]
        lw $t0,0($t1)           # Load addr from JmpTable

        jr $t0                  # Jump to correct code

L0:     add $s0,$s3,$s4         #  f = i + j
        j  Exit-Switch

L1:     add $s0,$s1,$s2         # f = g + h
        j  Exit-Switch

...etc...

Exit-Switch:
```

Basic Digital Logic

Integrated circuits (i.e., chips) in modern computers are made of digital circuits.

Some important physical components of digital circuits are:

1. Logic gates
   - implement boolean algebra/logic
   - inputs and outputs are electrical signals
2. Wires
   - connect combinations of gates
   - conduct electricity
3. Latches and flip-flops
   - used to temporarily hold a binary value
   - actually a combination of gates (see Appendix B)
4. Clock
   - a periodic electrical pulse
   - important for latches and flip-flops

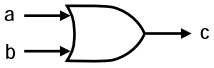
Gates implement boolean logic: AND gate, OR gate, invertor, multiplexor

Figure 4.8, pg. 231

1. AND gate ($c = a \cdot b$)

a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate ($c = a + b$)

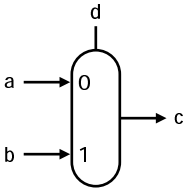


a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)

a	$c = \bar{a}$
0	1
1	0

4. Multiplexor
(if d = = 0, c = a;
else c = b)

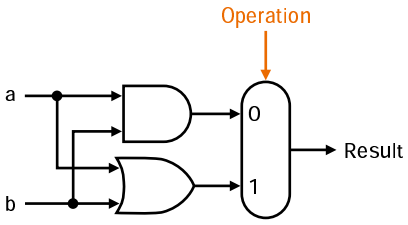


d	c
0	a
1	b

A multiplexor is really a "selector" gate.

How is a multiplexor implemented? Answer: Using other gates.

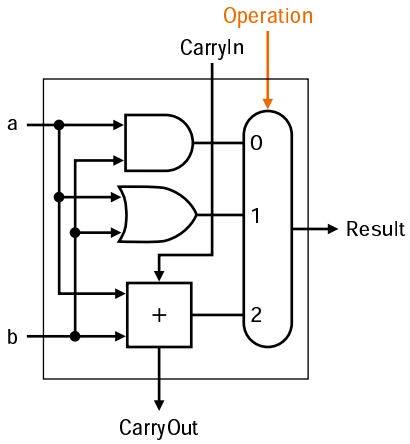
A combination of gates and multiplexors allows a digital circuit to be used for different purposes (i.e., computation) based on the input to the multiplexor (i.e., operation).



Gates can also be combined to perform arithmetic as well as boolean logic.

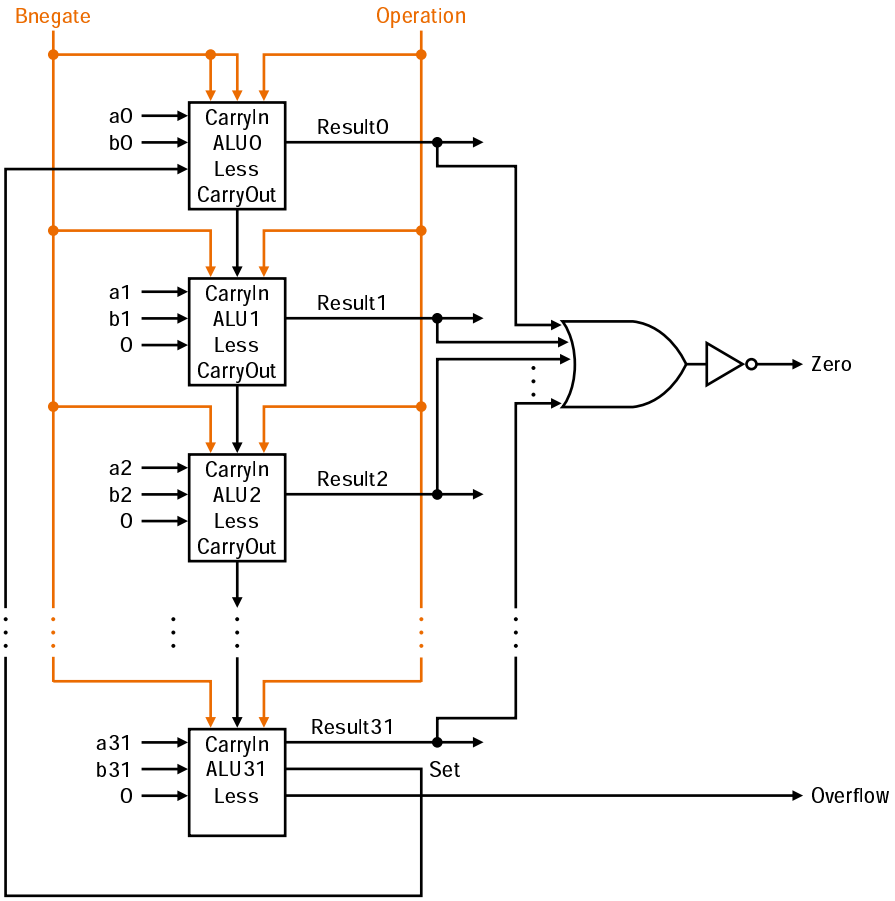
Together, such a circuit is called an arithmetic logic unit (ALU).

Figure 4.14, pg. 234 (A 1-bit ALU)



And 1-bit ALUs can be combined to form 32-bit ALUs.

Figure 4.19, pg. 240



The Processor: Datapath & Control

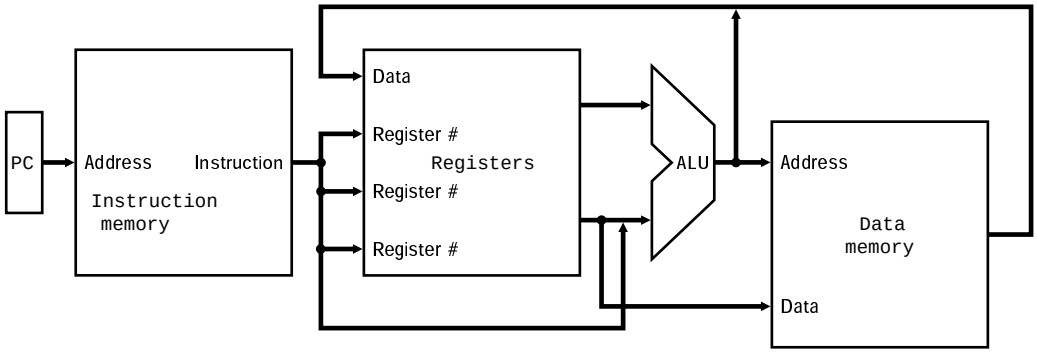
Recall that latches hold binary values.

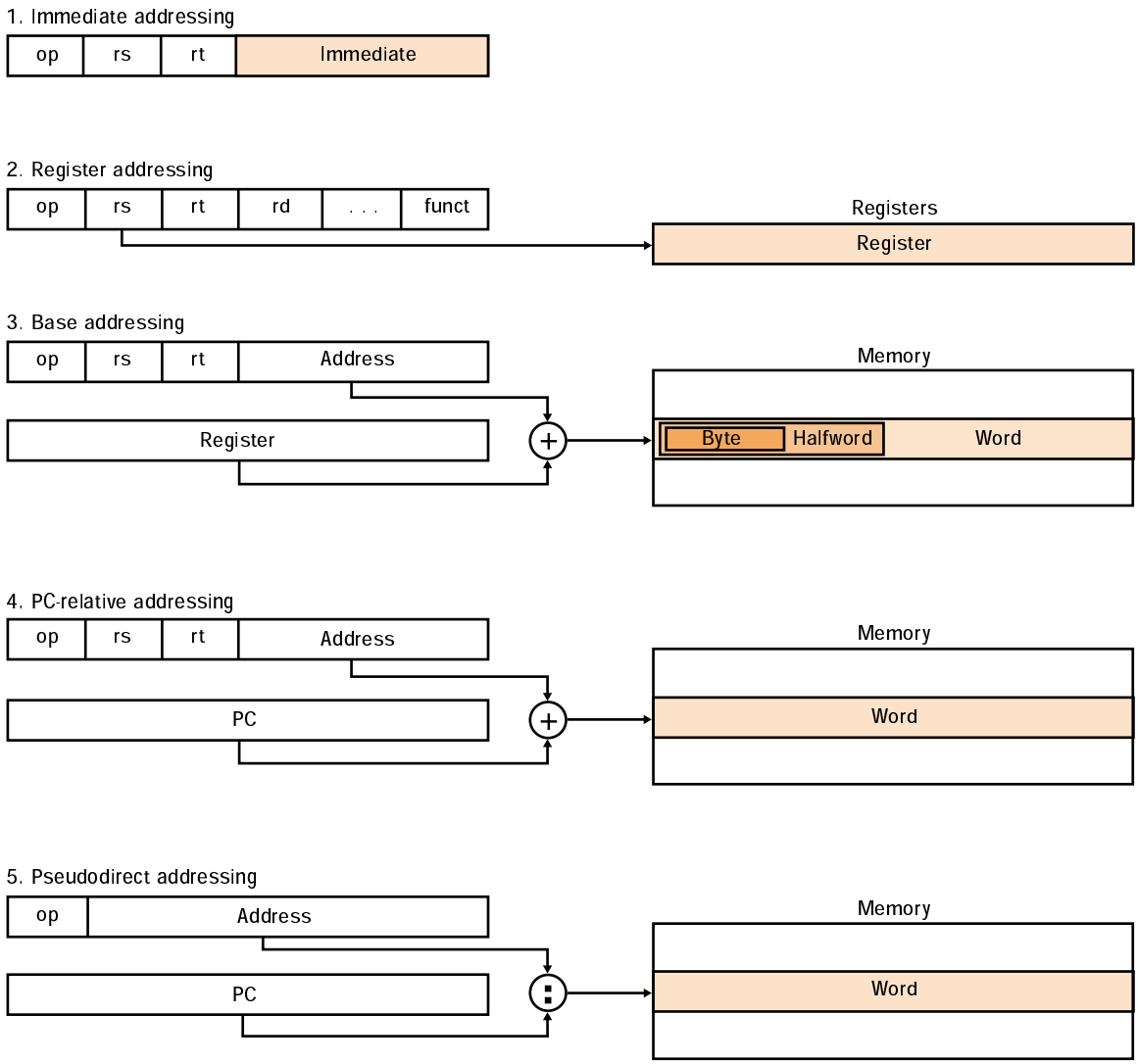
Therefore, they are "memory" components, which is what we need for registers, the program counter (PC), and memory.

The value of latches can change with every clock tick.

A high-level view of a CPU's digital circuit is (Figure 5.1, pg. 340) (next page).

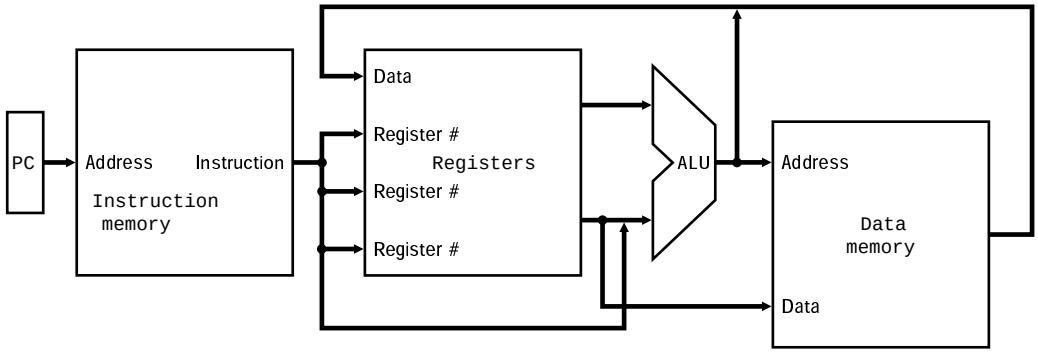
Boxes are memory components. Lines are wires. ALU is a combination of gates and wires.


PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
ALU
Address
Data memory
Data



Recall how instructions are encoded.

Fig. 3.17, pg. 152



What datapaths are used for the following?

1. `lw $s1,4($s2)`
   - rt = s1, rs = s2, offset = 4
2. `sw $s1,0($s2)`
   - rt = s1, rs = s2, offset = 0
3. `add $s1,$s2,$s3`
   - rd = s1, rs = s2, rt = s3, funct = `0x20`
4. `addi $s1,$s2,100`
   - rt = s1, rs = s2, imm = 100

Exceptions and Interrupts

Skip ahead to Chapter 5.6, pg. 410.

See also Appendix A.7, pg. A-32.

Problem:

How should we handle *abnormal* or *unpredictable* events?

Discussion:

We already use instructions like `beq` and `slt/bne` to change the *flow of control* for normal and predictable events.

With MIPS, an exception is any unexpected change in control flow.

What if the frequency of the events were low (but important when they occur) (e.g., errors) or if the events can occur at arbitrary times, not just within certain parts of the program (e.g., key is pressed).

Answer:

Modern CPUs use **exceptions** to handle abnormal internal or external events because the user:

1. does not want to constantly check for it (e.g., arithmetic overflow)
   - low frequency of these abnormal events
   - There are too many abnormal events to explicitly check for after each instruction.
2. does not know how to handle it (e.g., illegal or undefined instruction)
   - let the OS handle it
   - improve level of abstraction
3. should not handle it (i.e., privileged for the operating system)
   - let the OS handle it (i.e., system call)
   - for reasons of protection and security

Modern computers use **interrupts** to handle unpredictable *external* events because the user:

1. does not want to constantly check for it (e.g., key is pressed)
   - relatively low frequency of these events
   - can happen at any time
2. wants the event to be handled in a timely fashion (e.g., disk drive has the desired data)
   - want to avoid delay to avoid performance loss
   - want to avoid delay in handling in case data can be lost
   - can happen at any time

In many ways, handling an exception or interrupt is like a subroutine call:

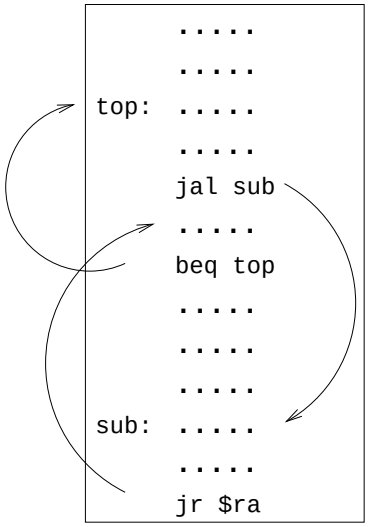
1. control flow is temporarily changed
2. control flow returns to original point when done

but:

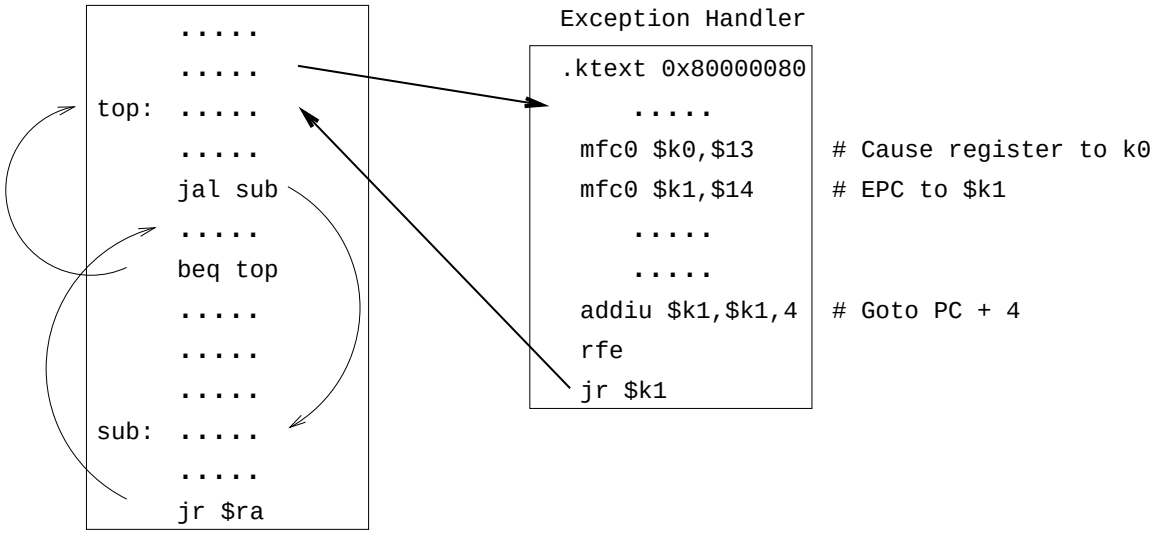
1. instead of a subroutine we have an *exception handler*, supplied by the operating system. For SPIM, the handler is at address `0x80000080`
2. we have `coprocessor 0` to manage exception-related information, including the *exception program counter* (EPC)
3. we have a *cause register*
4. we have a *status register*
5. we have `rfe` (in addition to `jr`) to return from an exception

Exceptions Revisited

Subroutines: Expected changes in control flow



Exception Handler: Unexpected changes in control flow

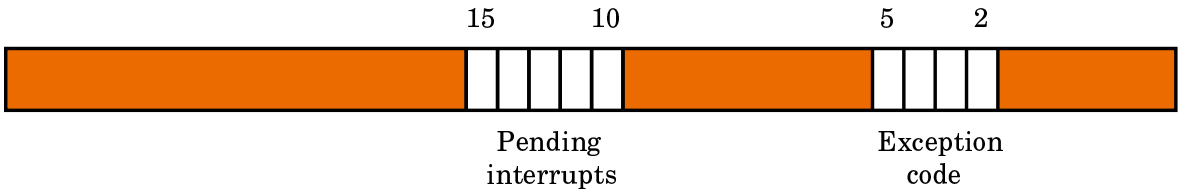


As part of exception handling, Coprocessor 0 must (see page A-32):

1. record the address of the offending instruction
   - EPC, aka register `$14`
   - NOTE: `$14` is not the same as the user's register 14, which is why we use `mfc0`
2. save the state of the CPU so that it can be restored with a `rfe` (*restore* from exception)
3. indicate the cause of the exception
   - cause register, aka register `$13`
4. make available the means to mask interrupts
   - status register, aka register `$12`
5. indicate the memory being accessed, if any, when exception occurred
   - BaddVAddr register, aka register `$8`
6. set to **kernel mode**, interrupts disabled

The Cause Register indicates why we are in the handler (pg. A-33).

- exception code = 0 $\Longrightarrow$ INT (external interrupt)
- exception code = 8 $\Longrightarrow$ SYSCALL (syscall exception)
- exception code = 10 $\Longrightarrow$ RI (reserved instruction exception)
- exception code = 12 $\Longrightarrow$ OVF (arithmetic overflow)

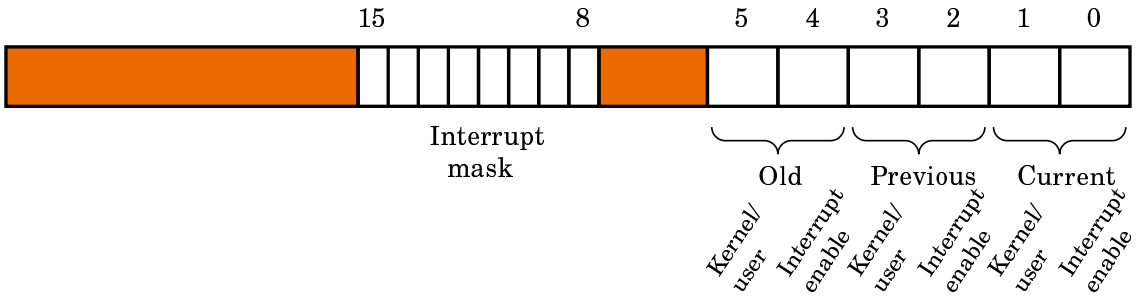


- can mask and shift the bits to get the exception code

Status Register:

Nesting exceptions and interrupts gets very tricky.

Among other things, an operating system would use the status register to mask interrupts (i.e., temporarily turn off or ignore selected interrupts)



A key concept is the notion of a **user state** vs. **system state**

Other names for system state include: supervisor state, privileged state, OS state, kernel mode

A modern operating system provides protection and security in this way

How are exceptions detected?

1. Interrupts from I/O devices: an electrical signal to the CPU itself from the motherboard
2. Undefined instruction: CPU control does not recognize the op code
3. Arithmetic overflow: ALU generates a signal out

(Review) Some important uses of exceptions include:

1. asking the operating system to perform some action
   - usually synchronous
   - system call (`syscall`)
   - trap
2. servicing an I/O device
   - usually asynchronous
   - keyboard is pressed
   - timer interrupt for context switch
3. dealing with low-frequency errors
   - usually synchronous
   - arithmetic overflow

`trap.handler`

The default exception handler is in file `trap.handler`, which is usually loaded when you start SPIM/XSPIM.

The default `trap.handler`:

1. prints out a simple message when an exception occurs
2. ignores interrupts

You have to modify this file in order:

1. to redefine the actions taken when an exception occurs
2. to handle interrupts

With SPIM, you can use your new exception handler by running with;

`spim -notrap -file` *your_filename*

Consider an annotated version of the default `trap.handler`.

```
# Paul:  This file contains the original trap.handler (no changes made
# Paul:         to kernel) and a main function, all in the same file.
# Paul:  Run with: spim -notrap -file trap.h.annotated

# Define the exception handling code.  This must go first!

        .kdata
__m1_:  .asciiz "  Exception "
__m2_:  .asciiz " occurred and ignored\n"
__e0_:  .asciiz "  [Interrupt] "
__e1_:  .asciiz ""
__e2_:  .asciiz ""
__e3_:  .asciiz ""
__e4_:  .asciiz "  [Unaligned address in inst/data fetch] "
__e5_:  .asciiz "  [Unaligned address in store] "
__e6_:  .asciiz "  [Bad address in text read] "
__e7_:  .asciiz "  [Bad address in data/stack read] "
__e8_:  .asciiz "  [Error in syscall] "
__e9_:  .asciiz "  [Breakpoint] "
__e10_: .asciiz "  [Reserved instruction] "
__e11_: .asciiz ""
__e12_: .asciiz "  [Arithmetic overflow] "
__e13_: .asciiz "  [Inexact floating point result] "
__e14_: .asciiz "  [Invalid floating point result] "
__e15_: .asciiz "  [Divide by 0] "
__e16_: .asciiz "  [Floating point overflow] "
__e17_: .asciiz "  [Floating point underflow] "

# Paul:  __excp is an array of pointers

__excp: .word __e0_,__e1_,__e2_,__e3_,__e4_,__e5_,__e6_,__e7_,__e8_,__e9_
        .word __e10_,__e11_,__e12_,__e13_,__e14_,__e15_,__e16_,__e17_

# Paul:  s1 and s2 are save locations for this non-re-entrant handler

s1:     .word 0
s2:     .word 0
```

```
        .ktext 0x80000080

# Paul:  .set noat allows us to use register $at without assembler complaining

        .set noat
        # Because we are running in the kernel, we can use $k0/$k1 without
        # saving their old values.

# Paul: Since we could have had an exception in the middle of a
# Paul: pseudo-instruction we have to save $at too.

        move $k1 $at    # Save $at
        .set at
        sw $v0 s1       # Not re-entrant and we can't trust $sp
        sw $a0 s2
        mfc0 $k0 $13    # Cause

# Paul: A pending interrupt of level 0 would have pattern
# Paul:          11              Bit positions
# Paul:          1098 7654 3210
# Paul:
# Paul:          0100 0000 0000
# Paul: 0x44   = 0000 0100 0100
# Paul: Therefore, any interrupt would be greater than 0x44

        sgt $v0 $k0 0x44 # ignore interrupt exceptions

# Paul:  This comment was changed on Nov. 29/00.
# Paul:  Note that you should NOT do a PC += 4 for an interrupt, but
# Paul:    that's what this code (erroneously) does.
# Paul:  Branching to "ret" avoids clearing of Cause reg

        bgtz $v0 ret

# Paul:  Next instruction is a NOP, for the branch delay slot

        addu $0 $0 0
```

```
        li $v0 4        # syscall 4 (print_str)
        la $a0 __m1_
        syscall
        li $v0 1        # syscall 1 (print_int)
        srl $a0 $k0 2   # shift Cause reg
        syscall
        li $v0 4        # syscall 4 (print_str)
        lw $a0 __excp($k0)

# Paul: Unlike a real MIPS CPU, SPIM handles syscalls differently.
# Paul: Normally, you would not see a syscall inside the exception handler.

        syscall

# Paul:          11              Bit positions
# Paul:          1098 7654 3210
# Paul:
# Paul:   0x18 = 0000 0001 1000 => exception code = 0110
# Paul:   0110 = 6 = IBUS = bus error on instruction fetch

        bne $k0 0x18 ok_pc # Bad PC requires special checks
        mfc0 $a0, $14   # EPC
        and $a0, $a0, 0x3 # Is EPC word-aligned?
        beq $a0, 0, ok_pc
        li $v0 10       # Exit on really bad PC (out of text)
        syscall

ok_pc:
        li $v0 4        # syscall 4 (print_str)
        la $a0 __m2_
        syscall
        mtc0 $0, $13    # Clear Cause register
ret:    lw $v0 s1
        lw $a0 s2
        mfc0 $k0 $14    # EPC
        .set noat
        move $at $k1    # Restore $at
        .set at
        rfe             # Return from exception handler
        addiu $k0 $k0 4 # Return to next instruction
        jr $k0
```

The default `trap.handler` also contains the `__start` code that calls your `main`. We must include our `main` in the same file if we use a custom exception handler.

```
# Standard startup code.  Invoke the routine main with no arguments.

        .text
        .globl __start
__start:
        lw $a0, 0($sp)  # argc
        addiu $a1, $sp, 4 # argv
        addiu $a2, $a1, 4 # envp
        sll $v0, $a0, 2
        addu $a2, $a2, $v0
        jal main
        li $v0 10
        syscall         # syscall 10 (exit)


# Run with: spim -notrap -file trap.h.annotated
        .globl main
main:
        li $v0, 4       # syscall 4 (print_str)
        la $a0, allok
        syscall

        jr $ra

        .data
allok:  .asciiz "All is ok\n"
```

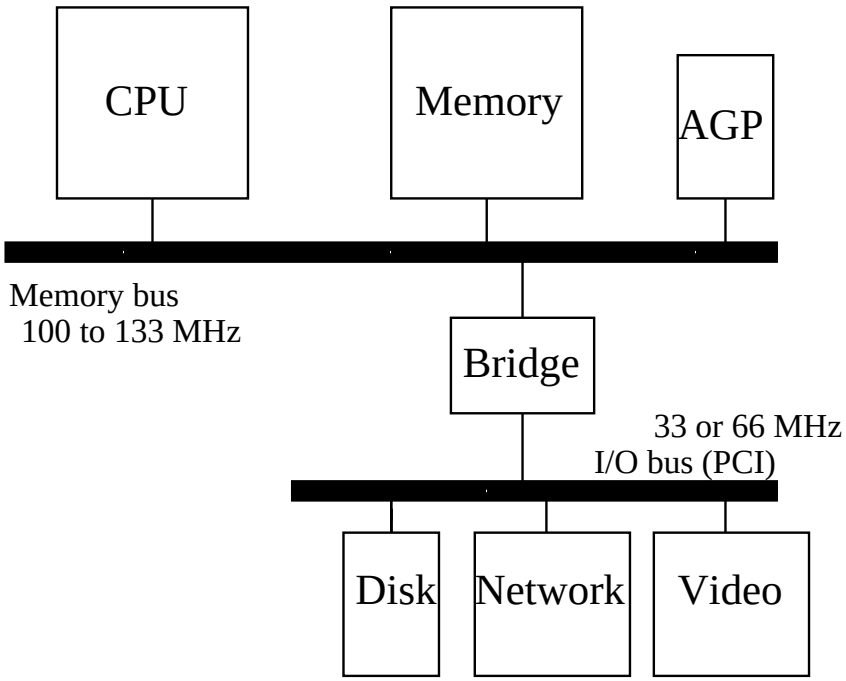
The above file is available at:

`http://www.cs.ualberta.ca/~paullu/C229/trap.h.annotated`

Input and Output

Without peripherals for input and output (I/O), a computer would not be very useful.

How would we program in C, play video games, or surf the Web without I/O devices?



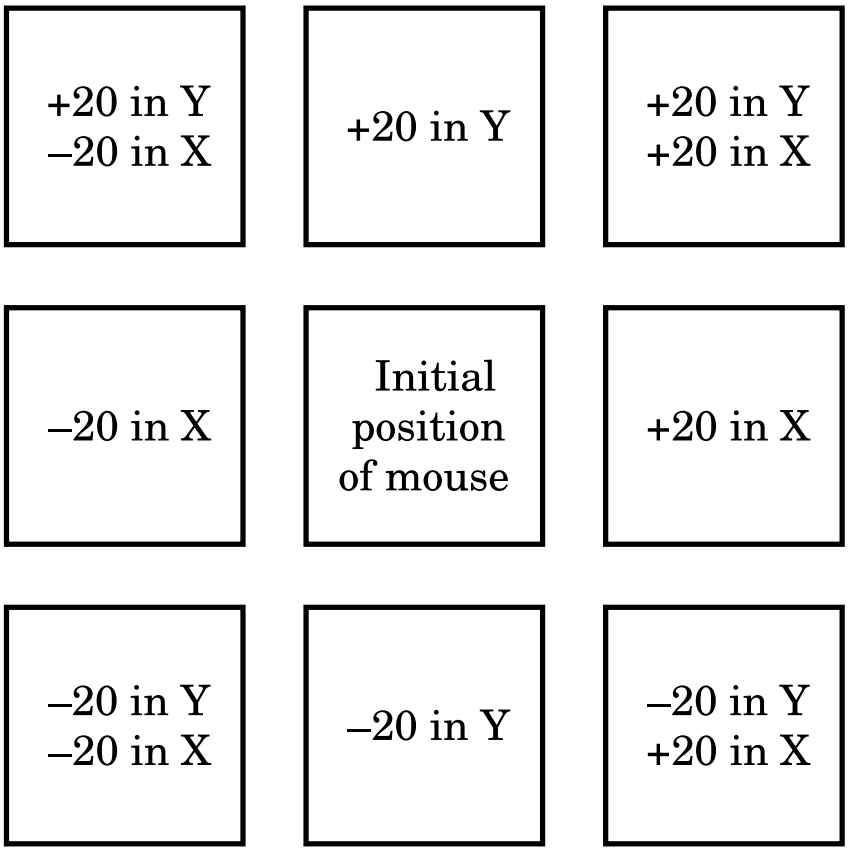
The textbook lists input and output as 2 of the 5 classic components of a computer.

There are a many kinds of I/O devices, each with different speeds (e.g., bytes/second of data transfer) (Figure 8.2 with updates).

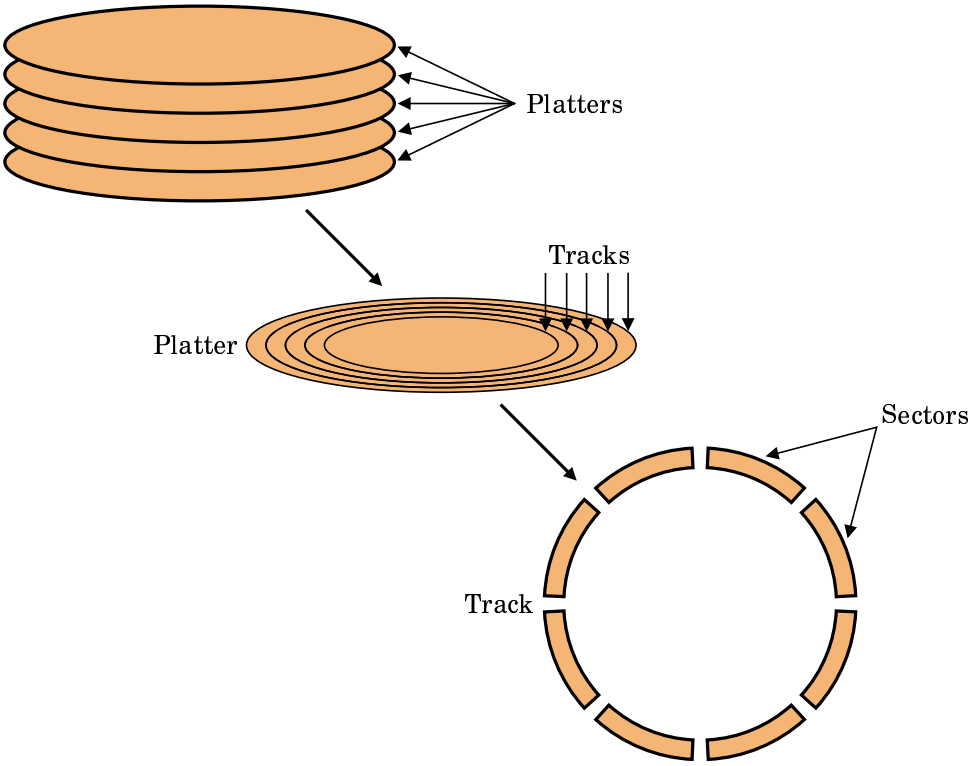
Device	Purpose	Data rate (KB/s)
Keyboard	input	0.01
Mouse	input	0.02
Voice input	input	0.02
Voice output	output	0.60
Laser printer	output	100 to 200
Graphics display	output	30,000 to 60,000
Network/LAN	both	10,000
Magnetic disk	storage	10,000

Note that CPUs run at 800 MHz to 1.4 GHz today (i.e., CPUs are much faster than I/O devices).

A mouse (Figure 8.3):



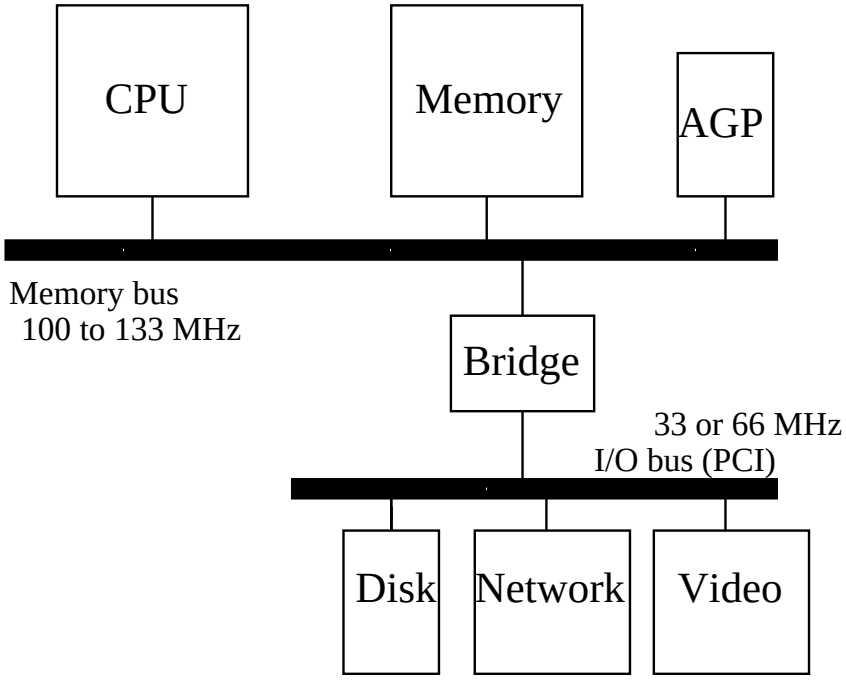
Hard (magnetic) disk (Figure 8.4):



To make I/O work, we need:

1. a way to connect the I/O devices to the processor and memory
   - E.g., commodity PCI I/O bus and bridge to memory bus
2. a way for the CPU to control the devices and transfer data
   - E.g., I/O instructions
   - E.g., coprocessors and direct memory access (DMA)
   - E.g., device registers: control and data registers
3. a convenient application programmer's interface (API)
   - E.g., `read()`, `write()`, `printf()`, `scanf()` in Unix
   - E.g., `syscall` in SPIM/XSPIM

Connecting I/O Devices



1. Commodity I/O buses, like PCI, make it cheaper to design and build I/O devices.
2. Memory bus can get faster independently.
3. Accelerated Graphics Port (AGP) for graphics: some special-purpose applications may want higher performance.
4. Moving data across 2 buses and a bridge chip may be slow.

Controlling I/O Devices

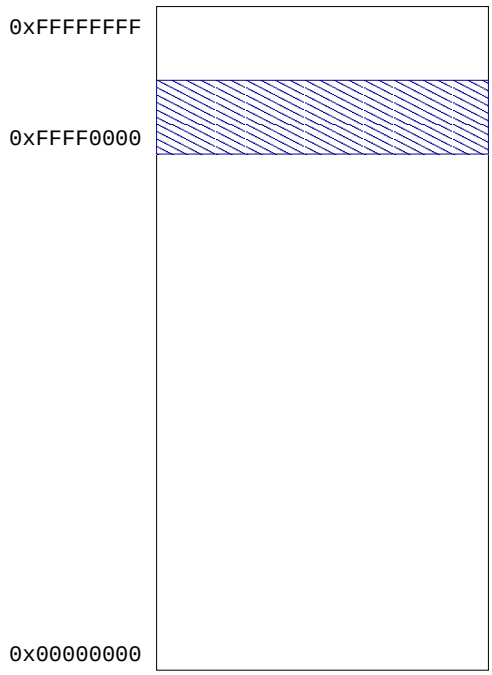
The CPU is "in command" of the computer.

Historically, the CPU controls I/O devices by either:

1. a CPU with special-purpose instructions (e.g., `OUTPUT D3,DISK`)
2. memory-mapped I/O: I/O devices are mapped to reserved portions of the address space

Memory-mapped I/O is the more common of the two options these days.

Memory-Mapped I/O



1. Selected addresses do not refer to RAM memory. Instead, the addresses are mapped to special "registers" on the I/O device.
2. Regular data movement instructions (e.g., `lw`, `sw`) control the I/O device and transfer data.
3. The memory controller hardware (and I/O bus hardware) redirects accesses (loads/stores) to these memory locations to the I/O device itself.

Peripheral Registers

- Peripheral devices are visible via collections of "registers" accessible as "special" memory locations.
- Unlike CPU registers, these "registers" are not for storing or computing with data.
- Rather, the action **and** value of data moved in and out of the "registers" have **side effects**: initializing a device, starting or stopping an I/O operation, changing the state of the device, clicking a speaker, etc.

  Compare this with the Status register in MIPS exception handling and turning on/off interrupts.
- The word "special" means that they are not really memory locations and they do not behave like ones.

Such a register may be in fact:

1. a data buffer storing an outgoing or incoming character

   To write an array of bytes, one may have to write each byte to the **same** address.

2. a control register for selecting/changing the parameters of the device

3. a command register (storing a value there may result in a specific operation being performed on the device)

4. an address register containing the address of a memory buffer taking part in a data transfer

Processor vs. I/O Speed Mismatch

(Adapted from Dr. Patterson's notes.)

A 1000 MHz (1 GHz) CPU can execute 1 billion load/stores per second.

$\Longrightarrow$ about 4,000,000 KB/s data rate (assuming the buses are fast enough)

But, I/O devices vary from 0.01 KB/s to 60,000 KB/s

$\Longrightarrow$ CPU is **much** faster than I/O

Problems:

1. For input, the device may not be ready to send as fast as CPU wants data
2. For output, the device may not able to receive data as fast as CPU wants to send it

The CPU and I/O device do not share a common clock, and the CPU is so much faster.

Therefore,

1. If the CPU sends data faster than the I/O device can handle it, *data could be lost*
   - i.e., writing to the device's data register before previous value is handled
2. If the CPU reads data faster than the I/O device send handle it, *data could be repeated*
   - i.e., reading from the device's data register before it changes to new value

What is the solution?

Flow control aka **handshaking** aka **synchronization**

Literally, before *starting to* send/receive data, the CPU checks if it is OK to send/receive so as to not overwhelm the slower device.

Pseudo-code for synchronization:

```
for( all data )
{
  while( device not ready ) { /* poll */ };
  send/receive next data item;
}
```

A control register indicates whether the "device is ready" or not.

Data is sent to/received from a data register.

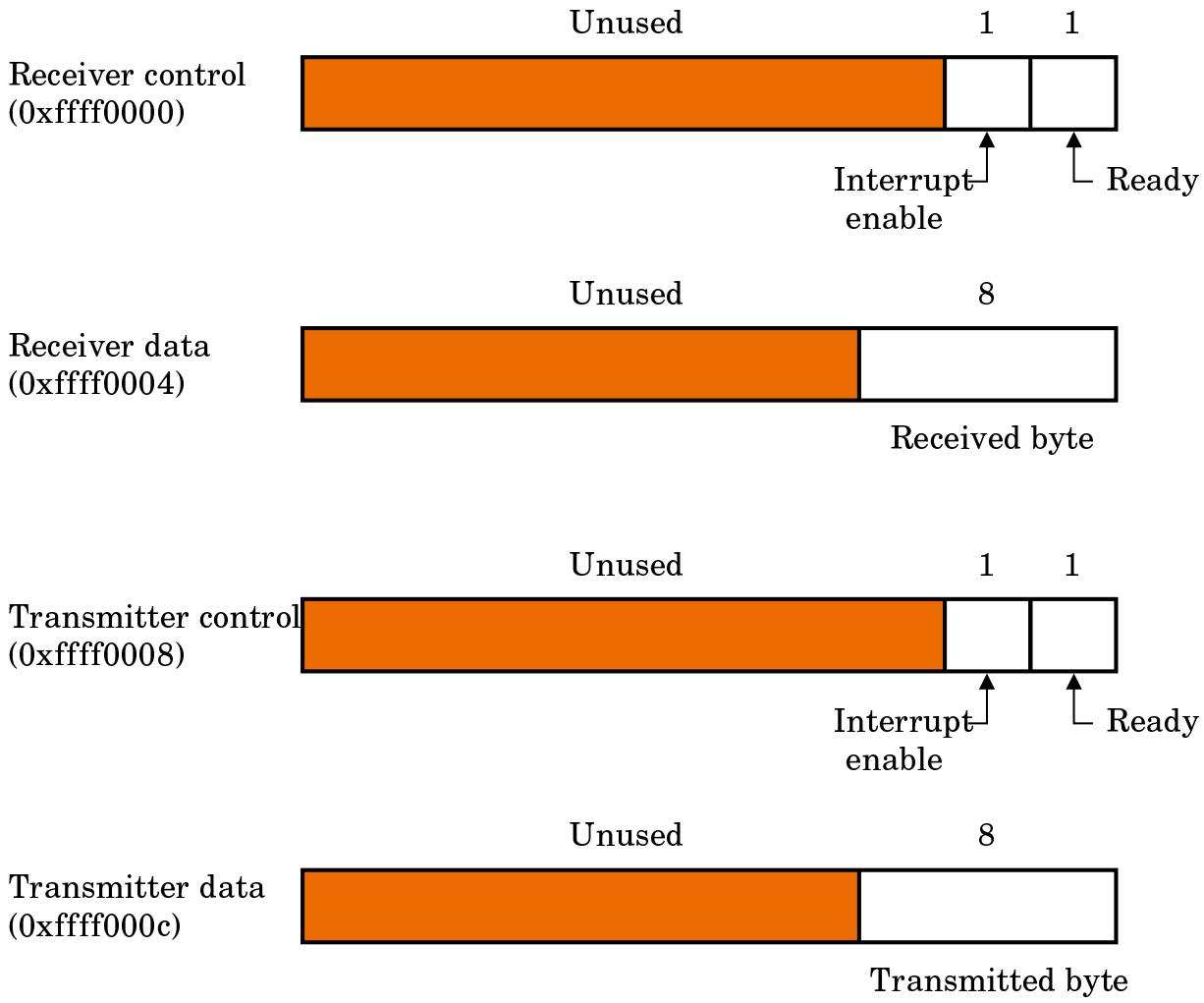
Normally, such a `while` loop would result in an infinite loop.

But, an external factor (i.e., the I/O device) intervenes to allow the loop to exit.

This is a simple example of synchronization (between CPU and peripheral), which is a major topic in operating systems.

SPIM/XSPIM only supports one memory-mapped I/O device: a character-based terminal

Figure A.15, pg. A-37:



Problem: Write a program to input a single character and echo it (twice).

```
# How to run:  spim -mapped_io -file ex10.echoinput.s
#         or:  xspim -mapped_io
        .text
        .globl main
main:
        li $t0,0xffff0000      # Base address for terminal
waitinput:
    # Polling loop to read input
        lw $t1, 0($t0)          # Recv control register
        andi $t1,$t1,0x1        # Ready for reading?
        beq $t1,$zero,waitinput # Branch if no

        lw $t2,4($t0)           # Yes, now get the byte

waitout:
    # Polling loop to wait for output to be ready
        lw $t1, 8($t0)          # Xmit control register
        andi $t1,$t1,0x1        # Ready for writing?
        beq $t1,$zero,waitout   # Branch if no

        sw $t2, 0xc($t0)        # Yes, now send the byte

waitout2:
    # Second polling loop for output
        lw $t1, 8($t0)          # Xmit control register
        andi $t1,$t1,0x1        # Ready for writing?
        beq $t1,$zero,waitout2  # Branch if no

        sw $t2, 0xc($t0)        # Yes, now send the byte
        j waitinput             # Loop forever...
```

What happens if we don't check for "ready for reading"? (Be quick with a Control-C.)

```
# How to run:  spim -mapped_io -file ex11.echoinput.s
#         or:  xspim -mapped_io
        .text
        .globl main
main:
        li $t0,0xffff0000      # Base address for terminal

waitinput:
    # Read the input whether ready or not
        lw $t2,4($t0)           # Yes, now get the byte

waitout:
    # Polling loop to wait for output to be ready
        lw $t1, 8($t0)          # Xmit control register
        andi $t1,$t1,0x1        # Ready for writing?
        beq $t1,$zero,waitout   # Branch if no

        sw $t2, 0xc($t0)        # Yes, now send the byte

waitout2:
    # Second polling loop for output
        lw $t1, 8($t0)          # Xmit control register
        andi $t1,$t1,0x1        # Ready for writing?
        beq $t1,$zero,waitout2  # Branch if no

        sw $t2, 0xc($t0)        # Yes, now send the byte
        j waitinput             # Loop forever...
```

What happens if we don't check for "read for writing" on the second output?

```
# How to run:  spim -mapped_io -file ex12.echoinput.s
#         or:  xspim -mapped_io
        .text
        .globl main
main:
        li $t0,0xffff0000      # Base address for terminal
waitinput:
    # Polling loop to read input
        lw $t1, 0($t0)          # Recv control register
        andi $t1,$t1,0x1        # Ready for reading?
        beq $t1,$zero,waitinput # Branch if no

        lw $t2,4($t0)           # Yes, now get the byte

waitout:
    # Polling loop to wait for output to be ready
        lw $t1, 8($t0)          # Xmit control register
        andi $t1,$t1,0x1        # Ready for writing?
        beq $t1,$zero,waitout   # Branch if no

        sw $t2, 0xc($t0)        # Yes, now send the byte

    # Send output again, ready or not
        sw $t2, 0xc($t0)        # Yes, now send the byte
        j waitinput             # Loop forever...
```

We still have problems:

1. The CPU spends a lot of time in the `waitinput`, `waitout`, and `waitout2` loops.
2. When there is a lot of data to be transferred to/from the device, the CPU is dedicated to that task.

Both are a "waste" of CPU cycles which could be used for something more useful, like computation.

What can we do?

1. Use interrupts to inform CPU when a device is ready, or otherwise needs attention.
2. Use special hardware to move data to/from device.

I/O Strategies

How do we actually transfer data in and out of an I/O device?

The choice of strategy depends on:

1. capability of the hardware device
2. the length or complexity of the I/O operation

Our three main strategies:

1. **Programmed I/O**: uses polling; CPU is dedicated to the task for the whole operation
2. **Interrupt-driven I/O**: CPU can do other computation; I/O device will interrupt if it needs attention
3. **DMA** (direct memory access): the I/O device can move data in and out of memory without the CPU's help; I/O device will interrupt if it needs attention

Programmed I/O

- Programmed I/O's strength is simplicity. Concurrency and (asynchronous) interrupt handling are hard for anything but trivial operations.
- Even efficient for very small data transfers; avoids interrupt latency.
- Again, its weakness is that the CPU must orchestrate the entire I/O operation; inefficient for large data transfers.
- A key feature of programmed I/O is **polling**: repeated checking for a given condition or event to occur (usually without some indication that it might have occurred)

Interrupt-driven I/O

Recall that an interrupt is an externally triggered exception.

Interrupt-driven I/O allows the I/O programmer to avoid polling, but it is more complicated and has the overhead of interrupt latency and setup in real life (but not SPIM).

There are three main parts:

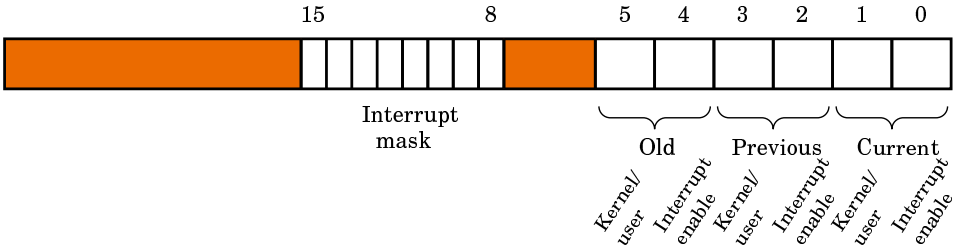
1. Initialization; enable interrupts
   - See the `.globl __start` part of `trap.handler`
2. Main program (that gets interrupted)
   - Whatever the program is supposed to be doing
3. Interrupt handler
   - The `.ktext 0x80000080` part of `trap.handler`

Enabling Interrupts

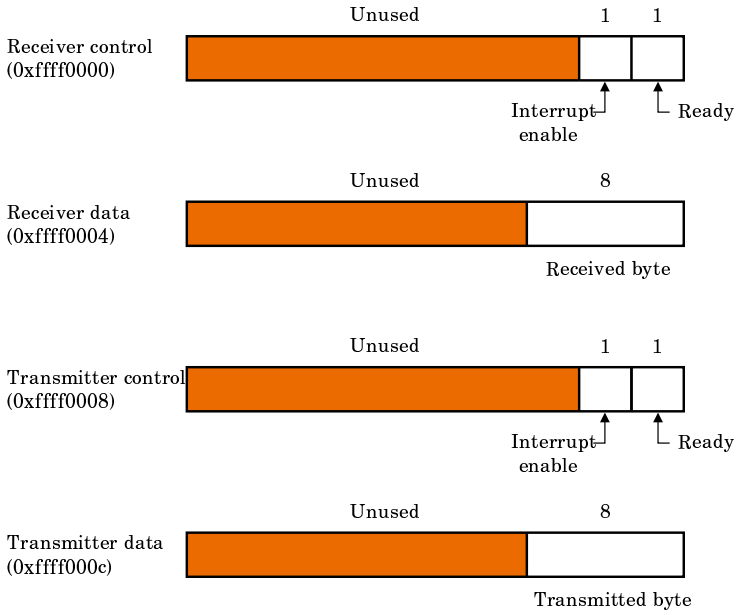
Interrupts are disabled by default.

To enable interrupts under SPIM/XSPIM, one must:

1. Enable "global" interrupts by setting bit 0 of the Status register (Figure A-13, pg. A-33).

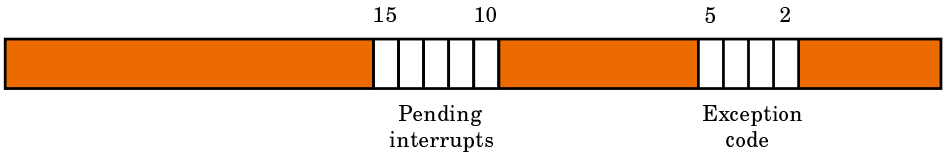


2. Individually enable interrupts for each desired I/O device (Figure A.15, pg. A-37)



Interrupt Handler

Recall that the interrupt handler is the same as the exception handler. Therefore, the handler must determine "why" it is being executed by examining the Cause register (Figure A-14, pg. A-33):



1. If the an exception has occurred, the Exception Code > 0 (See pg. A-33)
2. If an interrupt has occurred, the Exception Code $== 0$ and (according to the textbook):
   (a) If a keyboard (i.e., receiver) interrupt has occurred, then Pending Interrupt **bit 10** is set.
   (b) If a console/screen (i.e., transmitter) interrupt has occurred, then Pending Interrupt **bit 11** is set.

In general, there may be multiple devices sharing the same interrupt priority level. Therefore, one has to find out exactly which device of priority X caused the interrupt:

1. A more sophisticated interrupt system may have an interrupt device register which uniquely identifies the I/O device.
2. SPIM/XSPIM is much simpler/less intelligent: you have to check the Ready bits of the Receiver Control register and the Transmitter Control register (i.e., do a single poll).

   But, since SPIM/XSPIM only has 2 devices with different priority levels, you can avoid this check.

Even more generally, a real OS has to deal with many different interrupts, of different priorities, from different devices, and even nested interrupts. Very difficult to debug!!

Direct Memory Access (DMA) I/O

For complete **I/O operations**, the three I/O strategies are used in combination.

Interrupt-driven I/O avoids polling for **readiness**, but how is the **data transfer** handled?

Two main options:

1. Programmed I/O (PIO)
   - e.g., use PIO for both readiness and data transfer
   - Efficient for small amounts of data
2. Direct Memory Access (DMA)
   - e.g., use interrupts for readiness and DMA for data transfer
   - Efficient for large amounts of data
   - Requires extra hardware: a DMA controller

DMA Controller

The DMA controller is "smart" enough to move data and arbitrate for access to various buses, **independently of the CPU**.

Generally, it has one (or more) pairs of registers:

1. start address
2. end address

The DMA controller contains digital logic that is capable of the executing the following pseudocode:

```
current = start;
while( current <= end )
{
    move current to device;
    current++;
}
```

The DMA controller can interrupt the CPU (if enabled) when:

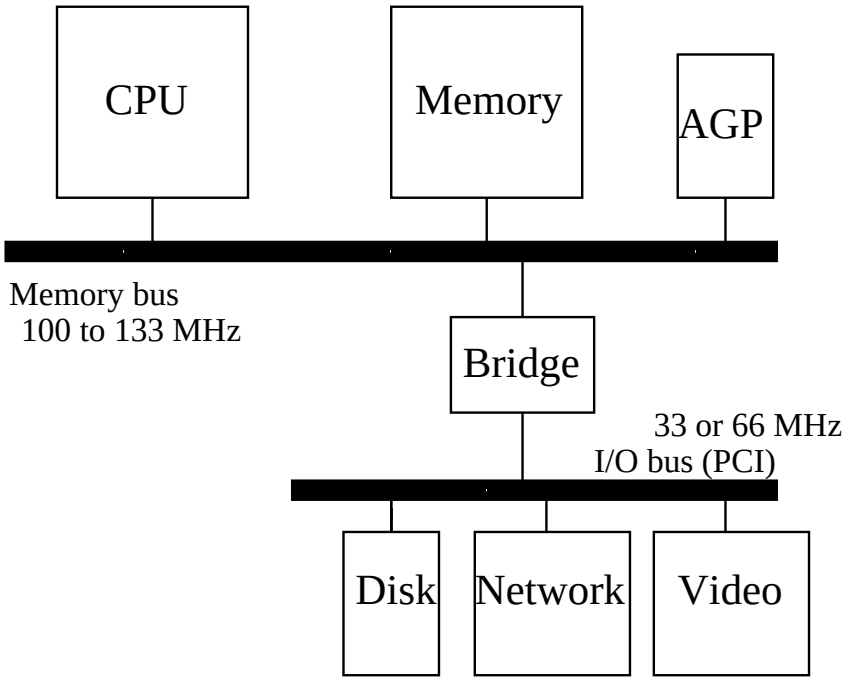
1. Ready to be used
2. Error condition has occurred
3. Device needs attention
4. When done a data transfer

$\Longrightarrow$ DMA controller has its own Status and Cause registers

NOTE:
The DMA controller is NOT a general-purpose CPU!

- Programmed I/O requires multiple trips across memory bus (wasting bandwidth)
- DMA controllers have configuration, startup, and interrupt overheads
- Programmable controllers on I/O devices (such as network interface cards) can help reduce the frequency of interrupts.

  Sometimes, these are called I/O Processors (IOP)



Advantages of DMA:

1. allows CPU to do other computation
   - concurrency
2. fewer bytes moved on memory bus than PIO
   - data does not go through CPU's registers
3. eliminate cache pollution

Disadvantages of DMA:

1. harder to program due to concurrency
2. contention for the memory bus
   - between CPU and DMA(s)
3. DMA overheads

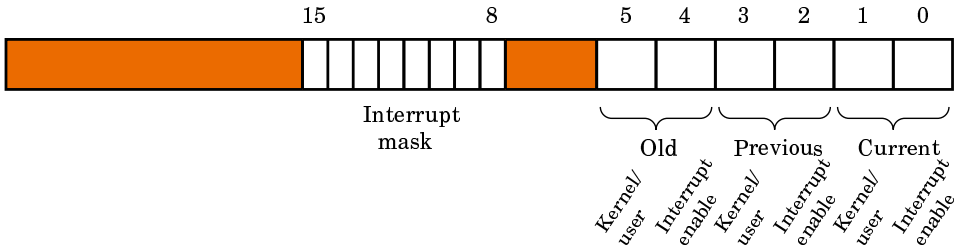
Again, the I/O strategies can be used in combination:

	Ready	Data Transfer	Done
Combo 1	PIO	PIO	
Combo 2	Interrupt	PIO	
Combo 3	PIO	DMA	Interrupt
Combo 4	Interrupt	DMA	Interrupt

- Combo 1 is good for "non-busy" devices and small data transfers
- Combo 2 is good for "busy" devices and small data transfers
- Combo 3 is good for "non-busy" devices and large data transfers
- Combo 4 is good for "busy" devices and large data transfers

Interrupt Masks

(Re: Slide 124, Enabling Interrupts) Enabling "global" interrupts requires 2 different steps:

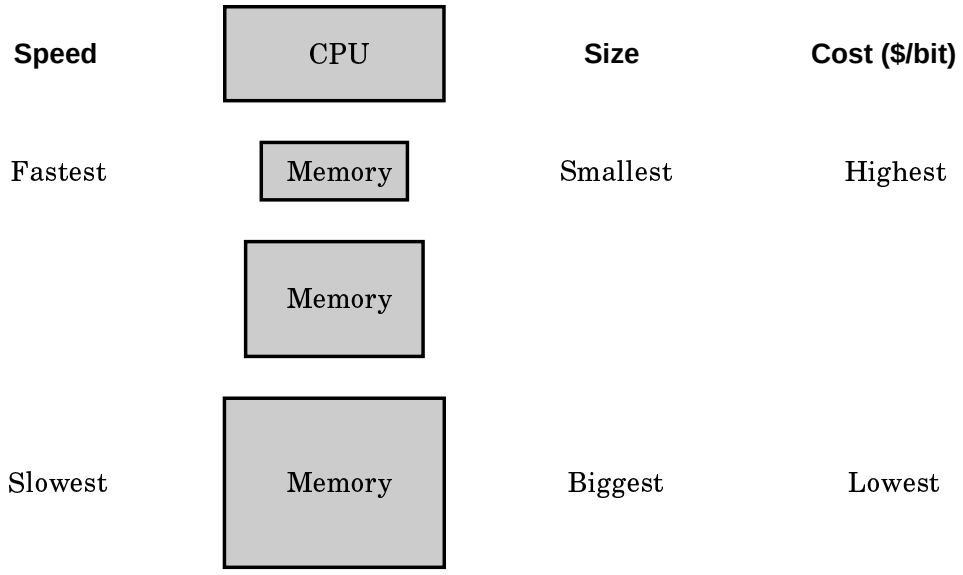


1. Setting bit 0 of the Status register (Figure A-13, pg. A-33).
2. Setting the appropriate Interrupt Mask (pg. A-33)
   (a) bit 8 for the keyboard
   (b) bit 9 for the console

Memory Hierarchy and Caches

A typical computer system has a variety of different data storage "devices."

Figure 7.1, pg. 542



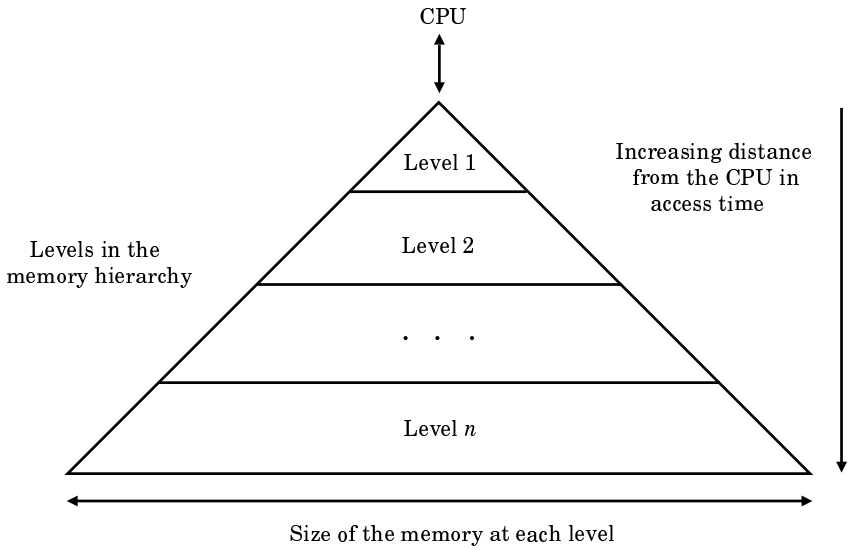
For example, in order of fastest to slowest (and most expensive to cheapest), we have:
CPU registers $\Longrightarrow$ level 1 cache $\Longrightarrow$ level 2 cache $\Longrightarrow$ main memory $\Longrightarrow$ hard disk

But, we cannot afford to keep all data in CPU registers.

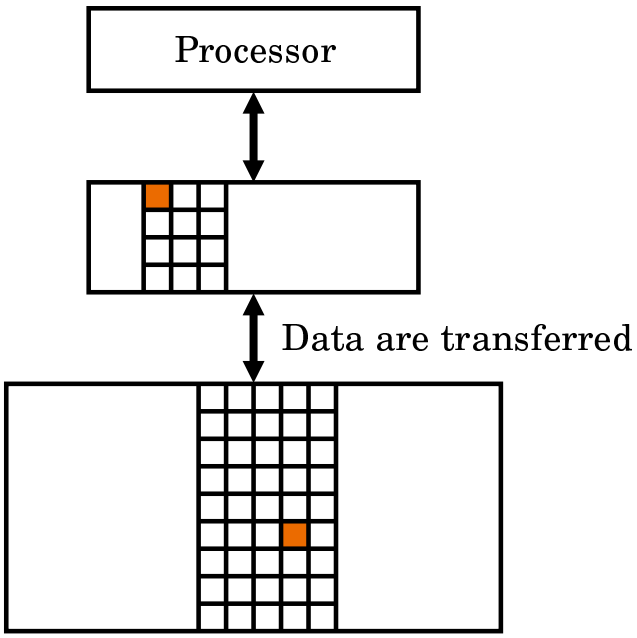
Therefore, we build a *memory hierarchy*.

Three important aspects of memory hierarchies:

1. "The pyramid": Inverse relationships between size, cost, and speed (Figure 7.3, pg. 544)

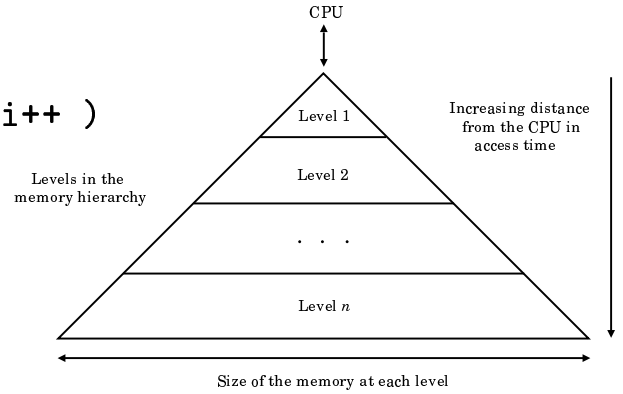


2. Movement of data between levels (Figure 7.2, pg. 543)



3. The principle of *locality of reference*: data access patterns can allow data to be reused in the higher levels of the memory hierarchy

```
int a[ 1024 ], i, j;

j = 0;
for( i = 0; i < 1024; i++ )
{
  j += a[ i ];
}
```



There are 2 main types of locality:

(a) *Temporal locality* (locality in time): if a data item is used, it may be used again in the "near future"
E.g., loop index variable, code for a loop

(b) *Spatial locality* (locality in (address) space): if a data item is used, items in neighbouring memory address locations may also be used
E.g., accessing array elements, instructions

```
int a[ 1024 ], i, j;

j = 0;
for( i = 0; i < 1024; i++ )
{
  j += a[ i ];
}
```

Note how variables `i` and `j`, together, exhibit both temporal and spatial locality of reference.

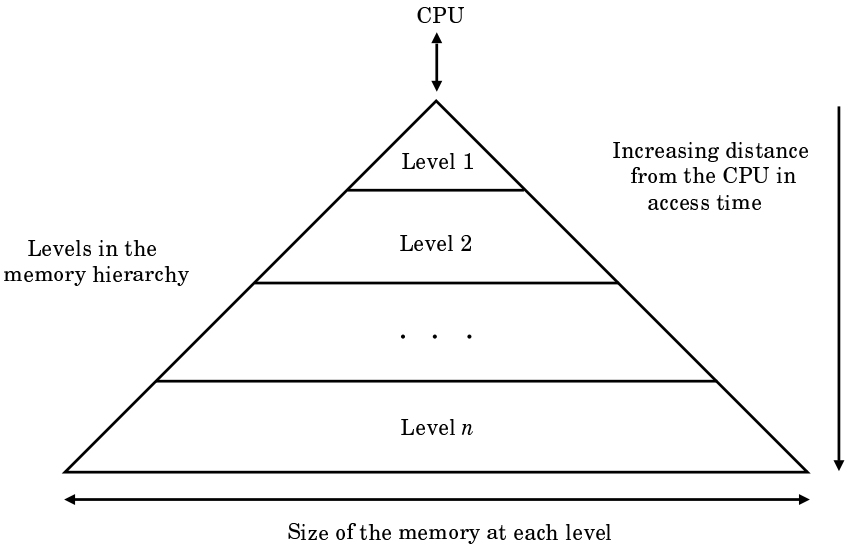
What kind of locality can we expect (or not expect) in typical data structures and algorithms?

1. stack
2. linked lists
3. C-style structures
4. mergesort
5. search a pointer-based tree

Why the pyramid relationship?

See Patterson's graph of the Processor-Memory Performance Gap, which grows at 50% / year.

$\Longrightarrow$ We add small amounts of fast memory, *caches*, between the processor and main memory



$\Longrightarrow$ if locality of reference is good, we move frequently used data into caches and access them at faster speeds than main memory.

$\Longrightarrow$ even if "fast" memory is small, we get most of the speed benefits at lower cost

$\Longrightarrow$ help close the Processor-Memory Performance Gap

The basic algorithm for caches is:

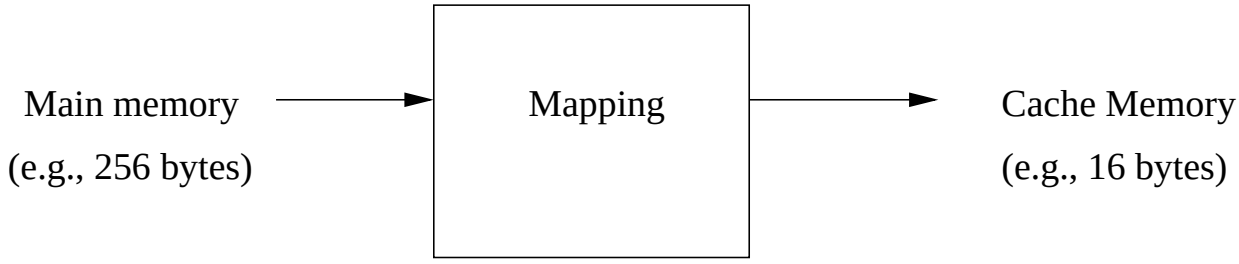
```
data memory-reference-with-cache( address )
{
    /* Invariant: Data is somewhere in memory hierarchy */

    if( data is not in cache )    /* Miss */
    {
        /* Miss penalty */
        if( cache is full )
            eject some data to make room
        move data from next lower level
    }

    /* Invariant: Data is now in cache */

    if( read access )             /* Hit, if not a miss */
    {
        return( data item at address )
    }
    else if( write access )
    {
        store data in memory hierarchy
        return NULL;
    }
}
```

Let $miss\_rate = (1 - hit\_rate)$

Then, Effective access time =
$hit\_cost \times hit\_rate + miss\_penalty(1 - hit\_rate)$

Cache Organization and Mapping

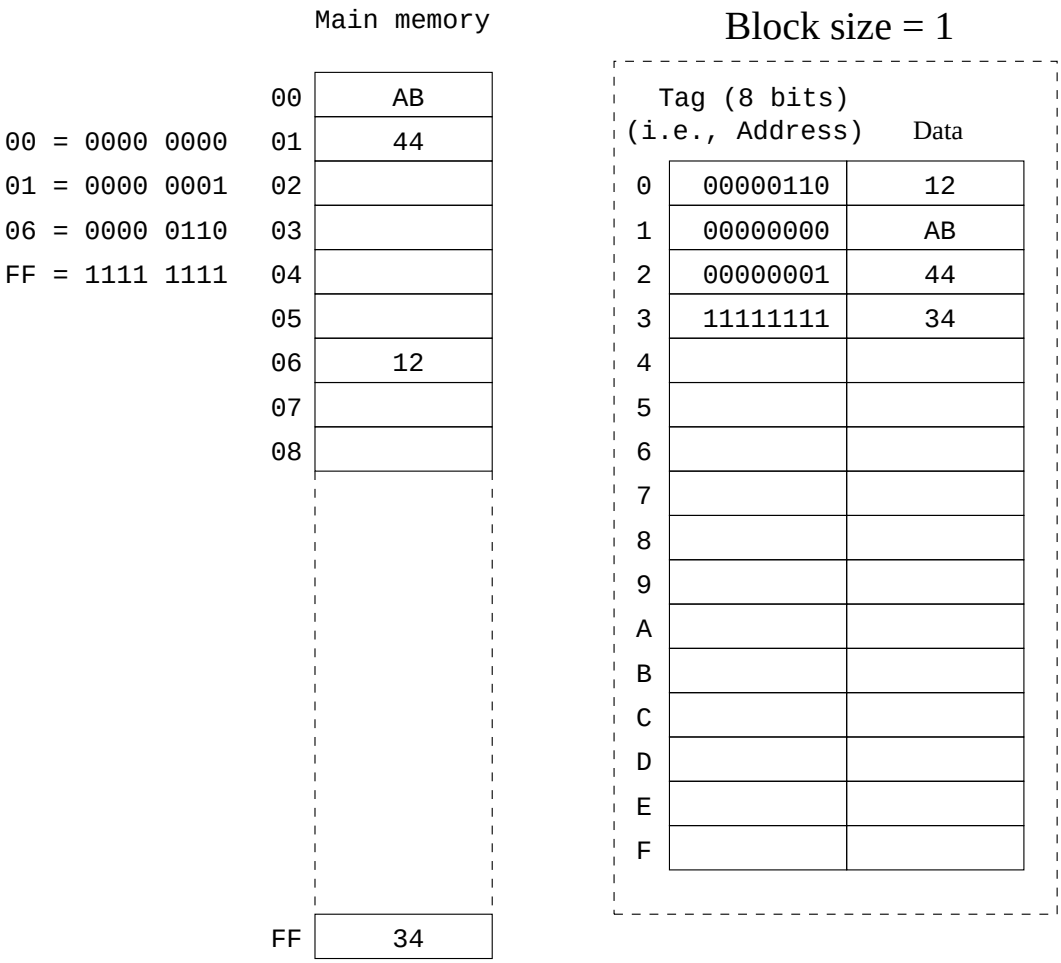
- Problem: Main memory is larger than cache memory. Yet, every byte of main memory has to have a possible storage location in cache.



- Solution: A mapping (or hashing) function from an *address* in main memory to a location in cache memory.
- How is cache memory organized to support an efficient mapping? What is the mapping/hashing function?

Fully Associative Cache

- Conceptually the easiest to understand, but are (often) the slowest organization in practice (e.g., high *cache_access_cost*)
- Fully associative mapping function ⇒ each **line** of cache can hold **any** data item from main memory.
- A **tag** indicates what data is in a given cache line.

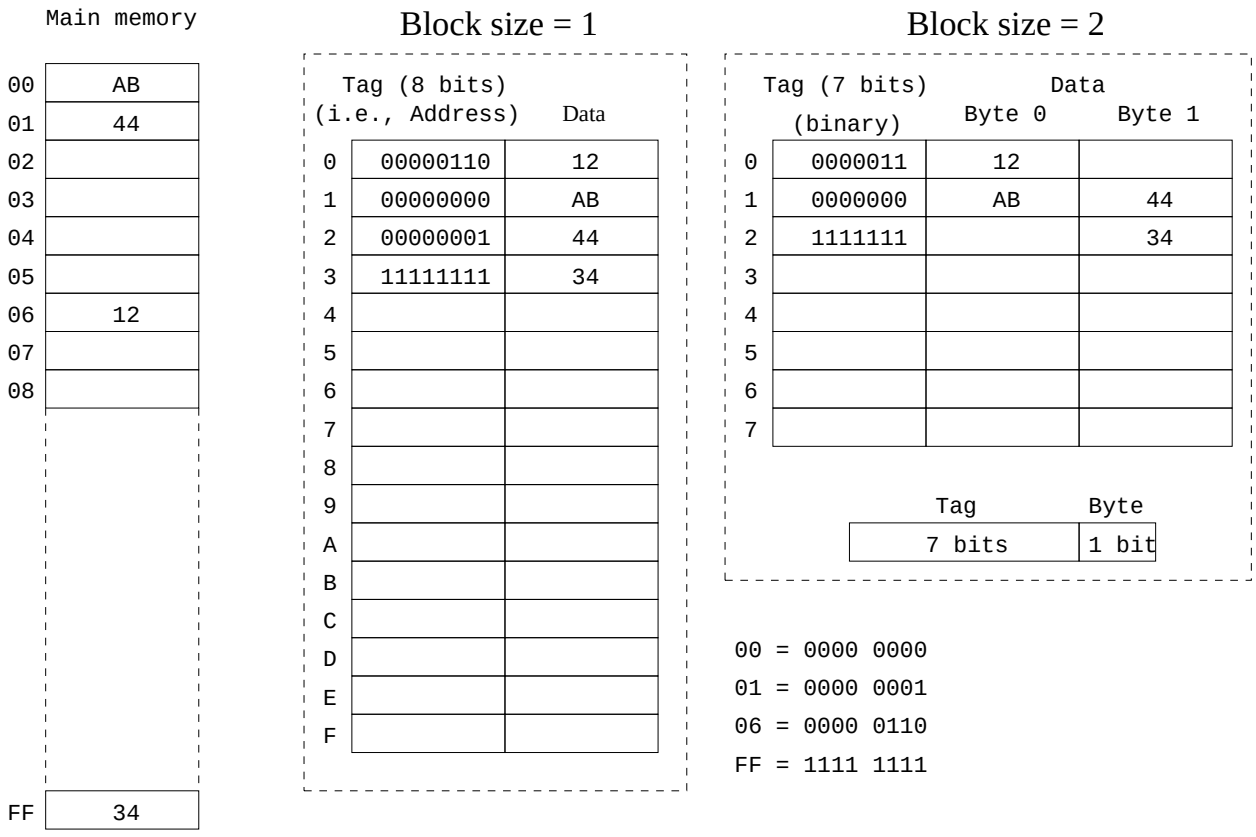


- To reduce the tag storage overhead and to move data to and from main memory more efficiently, we can group bytes of data into **blocks**.

  Blocks make it possible to exploit spatial locality.

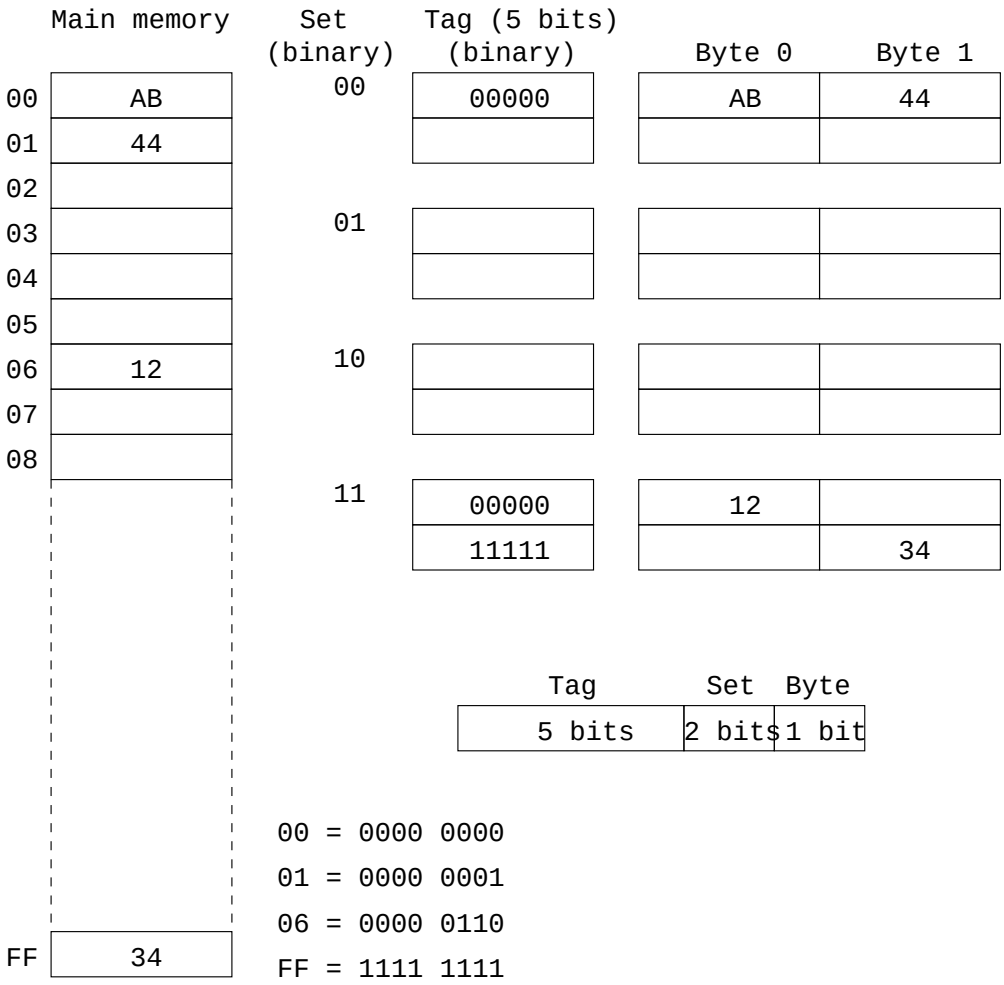
- A cache line is always the same size as a block.

- For block size 1, there are 128 tag bits ($= 8 \times 16$).
  For block size 2, there are 56 tag bits ($= 7 \times 8$).



2-way Set-Associative Cache

- Searching tags can be slow. We can reduce the number of tags to be searched by grouping cache lines into **sets**. Then, we only search the tags in a specific set. Sets are like buckets in a hash table.

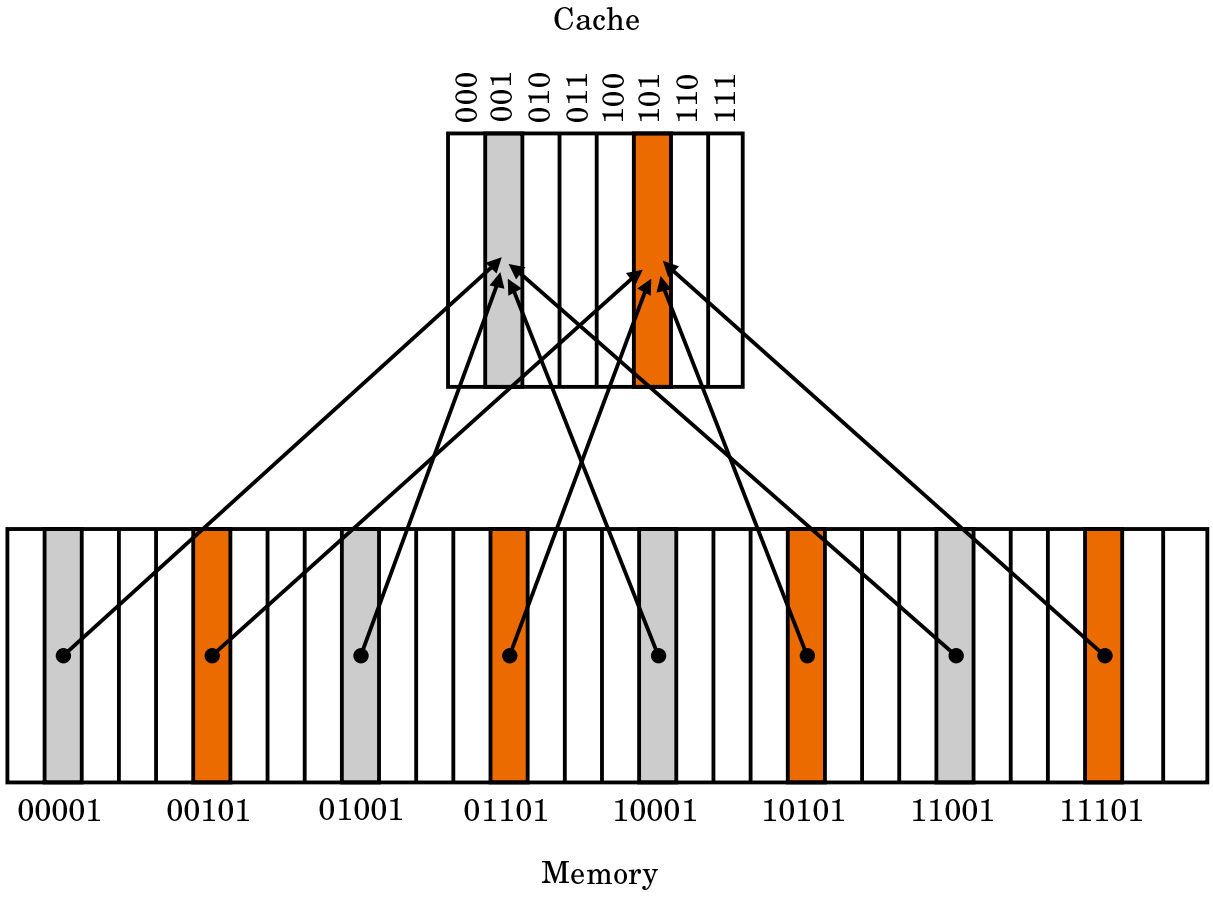


- If the set size is one (i.e., 1-way), then it is called a **direct-mapped cache**.

Direct-Mapped Cache

Figure 7.5, pg. 546

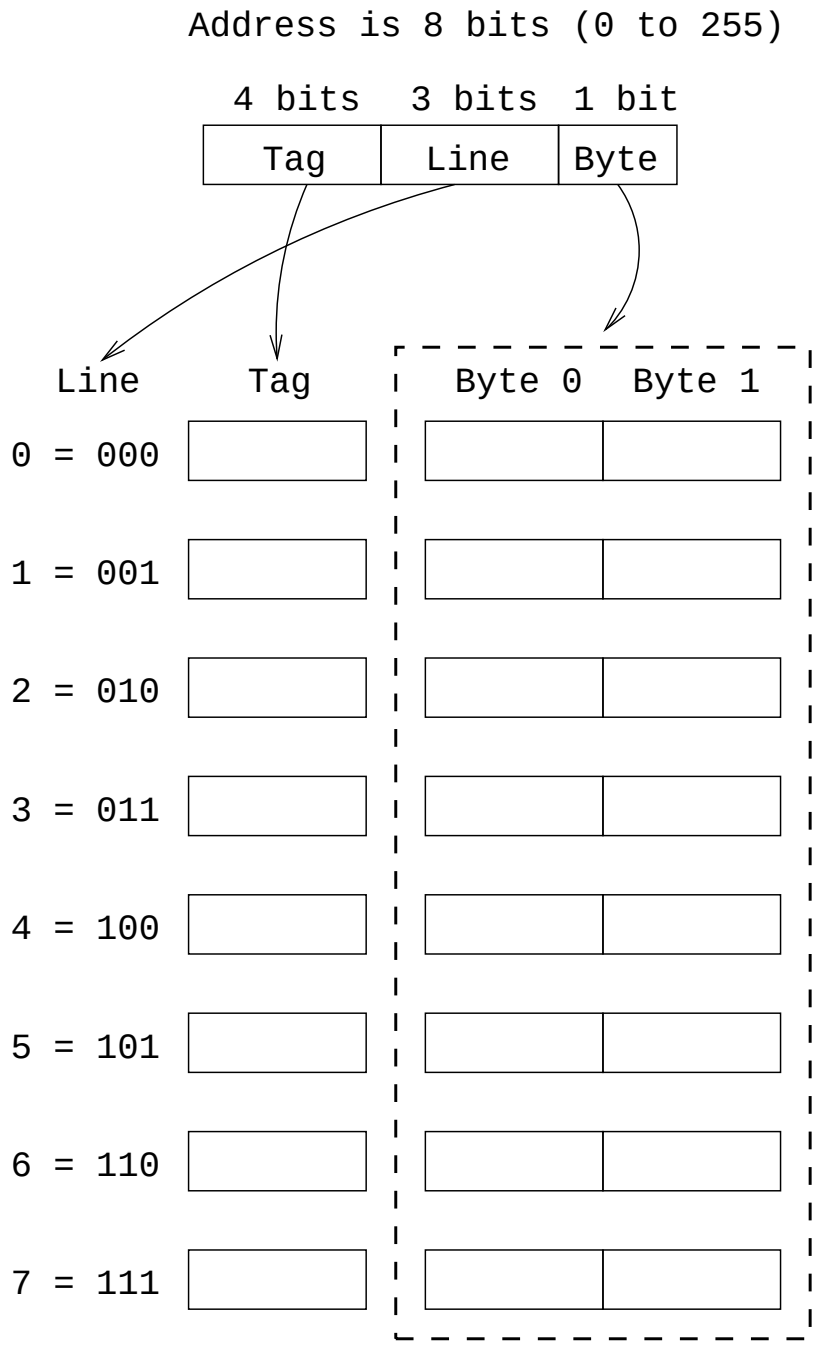
The addresses are expressed in binary. Assume block size of 1.



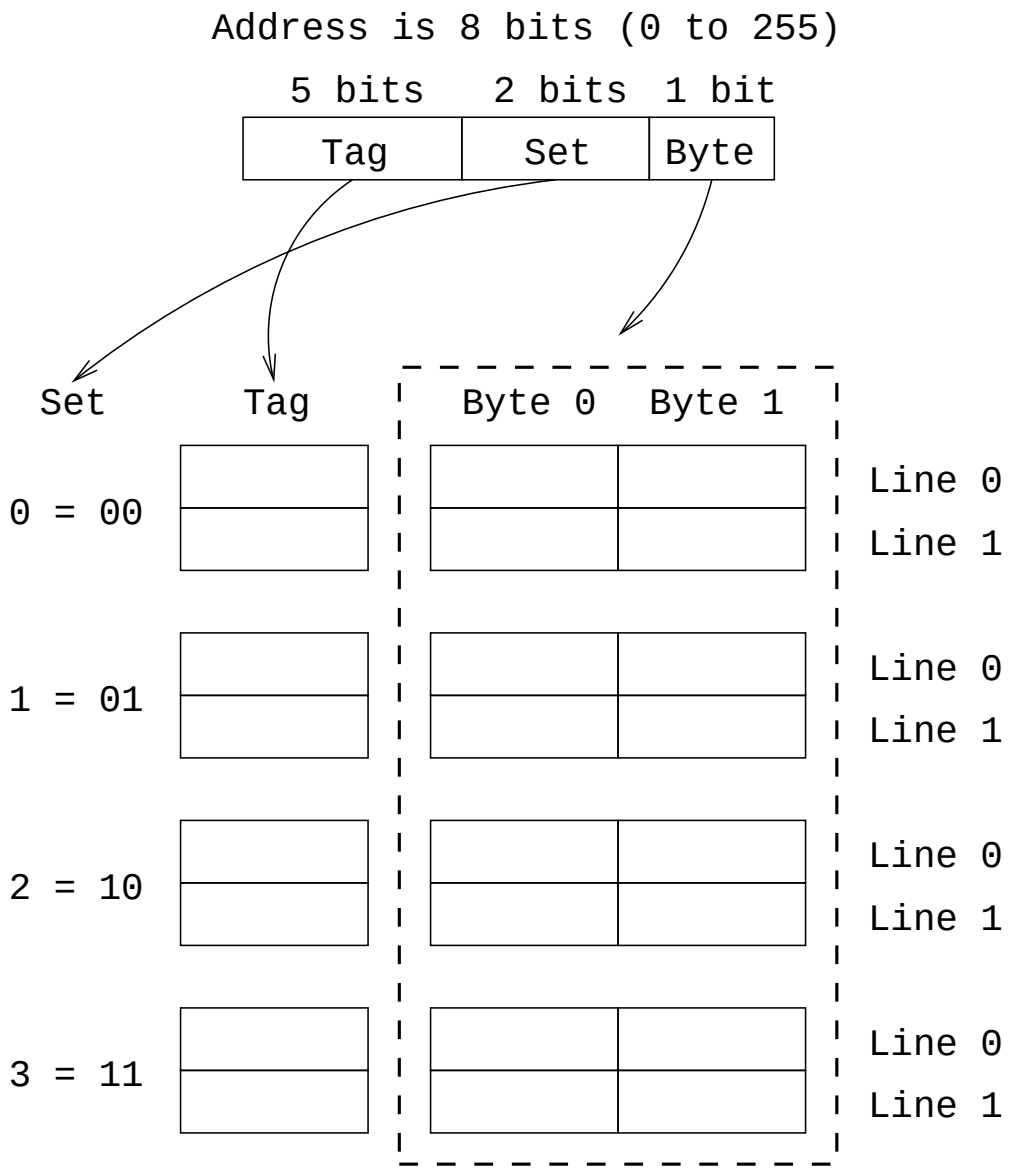
All addresses that end in `001` map to the same line in the cache.

All addresses that end in `101` map to the same line in the cache.

Direct-Mapped Cache: Template



2-way Set-Associative: Template



Example 1: Cache

Given an address space of 8 bits (0 to 255), a block size of 2 bytes, and a cache capable of holding 16 bytes (plus tag bits):

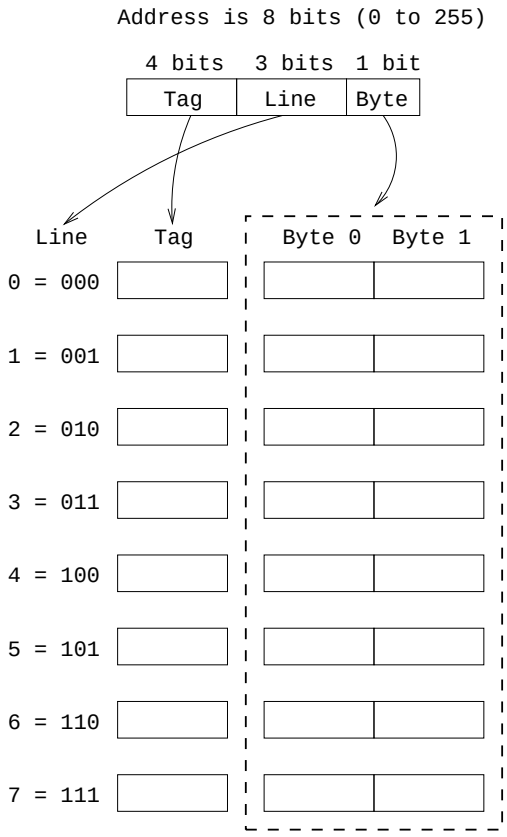
1. For a **direct-mapped** cache, how many lines are there in the cache? How many bits are required for the `Line` field? How many bits are in the `Tag` field? How many tag bits are required for the entire cache?
2. For a **2-way set-associative** cache, how many sets are there in the cache? How many lines in each set? How many bits are required for the `Set` field? How many bits are in the `Tag` field? How many tag bits are required for the entire cache?

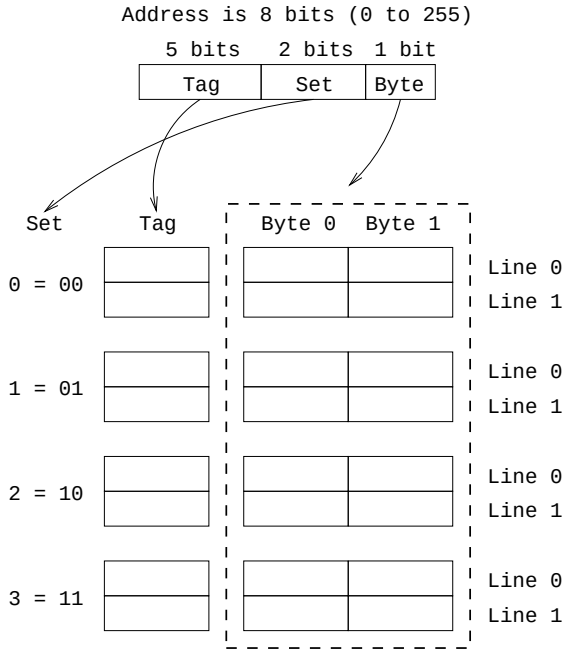
Now, what is the cache behaviour for the following address reference patterns:

1. 10, 20, 26, 36, 10, 20, 10 (Temporal locality)
2. 1, 2, 3, 4, 8, 9, 10 (Spatial locality)

Example 1: Cache (Answers)

Direct Mapped	2-way Set Associative
1. 8 bit address space 2. 16 bytes in cache 3. 2 bytes per block	1. 8 bit address space 2. 16 bytes in cache 3. 2 bytes per block
4. (3) ⇒ 1 bit for `Byte` 5. (2), (3) ⇒ 8 lines 6. (5) ⇒ 3 bits for `Line` 7. (4),(6) ⇒ 4 bits for `Tag` 8. (5),(7) ⇒ 32 bits for tags	4. (3) ⇒ 1 bit for `Byte` 5. (2),(3),2-way ⇒ 4 sets of 2 6. (5) ⇒ 2 bits for `Set` 7. (4),(6) ⇒ 5 bits for `Tag` 8. (5),(7) ⇒ 40 bits for tags





Example 1: Cache (Answers)

Reference Pattern: 10, 20, 26, 36, 10, 20, 10

(Temporal locality)

Addr.	Direct Mapped	2-way Set Assoc.
1. 10	$10 = 00001\mathbf{01}0_2$, Check Line $5 = 101_2$, Tag $= 0000_2 \Longrightarrow$ Miss	$10 = 00001\mathbf{0}10_2$ Check Set $1 = 01_2$ Tag $= 00001_2 \Longrightarrow$ Miss Use Line 0 of Set 1
2. 20	$20 = 00010\mathbf{10}0_2$ Check Line $2 = 010_2$, Tag $= 0001_2 \Longrightarrow$ Miss	$20 = 00010\mathbf{1}00_2$ Check Set $2 = 10_2$ Tag $= 00010_2 \Longrightarrow$ Miss Use Line 0 of Set 2
3. 26	$26 = 00011\mathbf{01}0_2$ Check Line $5 = 101_2$, Tag $= 0001_2 \Longrightarrow$ **Replace**	$26 = 00011\mathbf{0}10_2$ Check Set $1 = 01_2$ Tag $= 00011_2 \Longrightarrow$ Miss Use Line 1 of Set 1
4. 36	$36 = 00100\mathbf{10}0_2$ Check Line $2 = 010_2$, Tag $= 0010_2 \Longrightarrow$ **Replace**	$36 = 00100\mathbf{1}00_2$ Check Set $2 = 10_2$ Tag $= 00100_2 \Longrightarrow$ Miss Use Line 1 of Set 2
5. 10	$10 = 00001\mathbf{01}0_2$ Check Line $5 = 101_2$, Tag $= 0000_2 \Longrightarrow$ **Replace**	$10 = 00001\mathbf{0}10_2$ Check Set $1 = 01_2$ Tag $= 00001_2 \Longrightarrow$ **Hit** (Line 0)
6. 20	$20 = 00010\mathbf{10}0_2$ Check Line $2 = 010_2$, Tag $= 0001_2 \Longrightarrow$ **Replace**	$20 = 00010\mathbf{1}00_2$ Check Set $2 = 10_2$ Tag $= 00010_2 \Longrightarrow$ **Hit** (Line 0)
7. 10	$10 = 00001\mathbf{01}0_2$ Check Line $5 = 101_2$, Tag $= 0000_2 \Longrightarrow$ **Hit**	$10 = 00001\mathbf{0}10_2$ Check Set $1 = 01_2$ Tag $= 00001_2 \Longrightarrow$ **Hit** (Line 0)
	6 misses/replaces, 1 hit	4 misses, 3 hits

Reference Pattern: 1, 2, 3, 4, 8, 9, 10

(Spatial locality)

Addr.	Direct Mapped	2-way Set Assoc.
1. 1	$1 = 0000\mathbf{000}1_2$, Check Line $0 = 000_2$, Tag $= 0000_2 \Longrightarrow$ Miss	$1 = 00000\mathbf{00}1_2$ Check Set $0 = 00_2$ Tag $= 00000_2 \Longrightarrow$ Miss Use Line 0 of Set 0
2. 2	$2 = 0000\mathbf{001}0_2$ Check Line $1 = 001_2$, Tag $= 0000_2 \Longrightarrow$ Miss	$2 = 00000\mathbf{01}0_2$ Check Set $1 = 01_2$ Tag $= 00000_2 \Longrightarrow$ Miss Use Line 0 of Set 1
3. 3	$3 = 0000\mathbf{001}1_2$ Check Line $1 = 001_2$, Tag $= 0000_2 \Longrightarrow$ **Hit** Byte 1	$3 = 00000\mathbf{01}1_2$ Check Set $1 = 01_2$ Tag $= 00000_2 \Longrightarrow$ **Hit** Line 0, Byte 1
4. 4	$4 = 0000\mathbf{010}0_2$ Check Line $2 = 010_2$, Tag $= 0000_2 \Longrightarrow$ Miss	$4 = 00000\mathbf{10}0_2$ Check Set $2 = 10_2$ Tag $= 00000_2 \Longrightarrow$ Miss Use Line 0 of Set 2
5. 8	$8 = 0000\mathbf{100}0_2$ Check Line $4 = 100_2$, Tag $= 0000_2 \Longrightarrow$ Miss	$8 = 00001\mathbf{00}0_2$ Check Set $0 = 00_2$ Tag $= 00001_2 \Longrightarrow$ Miss Use Line 1 of Set 0
6. 9	$9 = 0000\mathbf{100}1_2$ Check Line $4 = 100_2$, Tag $= 0000_2 \Longrightarrow$ **Hit** Byte 1	$9 = 00001\mathbf{00}1_2$ Check Set $0 = 00_2$ Tag $= 00001_2 \Longrightarrow$ **Hit** Line 1 of Set 0
7. 10	$10 = 0000\mathbf{101}0_2$ Check Line $5 = 101_2$, Tag $= 0000_2 \Longrightarrow$ Miss	$10 = 00001\mathbf{01}0_2$ Check Set $1 = 01_2$ Tag $= 00001_2 \Longrightarrow$ Miss Use Line 1 of Set 1
	5 misses, 2 hits	5 misses, 2 hits

Example 2: Cache

Given an address space of 16 bits (0 to 65,535), a block size of 4 bytes, and a cache capable of holding 1,024 bytes (plus tag bits):

1. For a **direct-mapped** cache, how many lines are there in the cache? How many bits are required for the `Line` field? How many bits are in the `Tag` field? How many tag bits are required for the entire cache?
2. For a **2-way set-associative** cache, how many sets are there in the cache? How many lines in each set? How many bits are required for the `Set` field? How many bits are in the `Tag` field? How many tag bits are required for the entire cache?
3. For a **8-way set-associative** cache, how many sets are there in the cache? How many lines in each set? How many bits are required for the `Set` field? How many bits are in the `Tag` field? How many tag bits are required for the entire cache?

Example 2: Cache (Answers)

Given an address space of 16 bits (0 to 65,535), a block size of 4 bytes, and a cache capable of holding 1,024 bytes (plus tag bits). NOTE: 2 bits are required for the `Byte` field.

1. For a **direct-mapped** cache
   (a) There are 256 lines (= 1024/4)
   (b) 8 bits are required for the `Line` field ($2^8 = 256$)
   (c) 6 bits are in the `Tag` field ($= 16 - 2 - 8$)
   (d) 1,536 tag bits are required ($= 256 \times 6$)
2. For a **2-way set-associative** cache
   (a) There are 128 sets (= (1024/4)/2)
   (b) There are 2 lines in each set (i.e., 2-way)
   (c) 7 bits are required for the `Set` field ($2^7 = 128$)
   (d) 7 bits are in the `Tag` field ($= 16 - 2 - 7$)
   (e) 1,792 tag bits are required ($= 128 \times 2 \times 7$)
3. For a **8-way set-associative** cache
   (a) There are 32 sets (= (1024/4)/8)
   (b) There are 8 lines in each set (i.e., 8-way)
   (c) 5 bits are required for the `Set` field ($2^5 = 32$)
   (d) 9 bits are in the `Tag` field ($= 16 - 2 - 5$)
   (e) 2,304 tag bits are required ($= 32 \times 8 \times 9$)

Other Cache Issues

See Chapter 7.5.

1. **Block replacement policy**: Which block of data do we evict from the cache (i.e., replace) so that a new block of data can be brought into the cache?

   Basic idea: If we have to evict a block, choose the one that will most likely not be used again.

   (a) *Least Recently Used (LRU)*

   ⊕ further exploits locality and *working set* principles

   ⊖ hard to implement in hardware

   (b) *Approximation of LRU*

   ⊕ easier to implement in hardware

   (c) *Random*

   ⊕ easier to implement in hardware

2. **Write strategy**: When we write data into the cache, when do we update main memory?

   (a) *Write-through* (perhaps with write buffer)

   - update main memory on every write

   ⊕ lower miss penalty; faster to evict a block from cache

   ⊖ consumes memory bandwidth for each write

   (b) *Write-back* (also called copy-back)

   - update main memory on block replacement

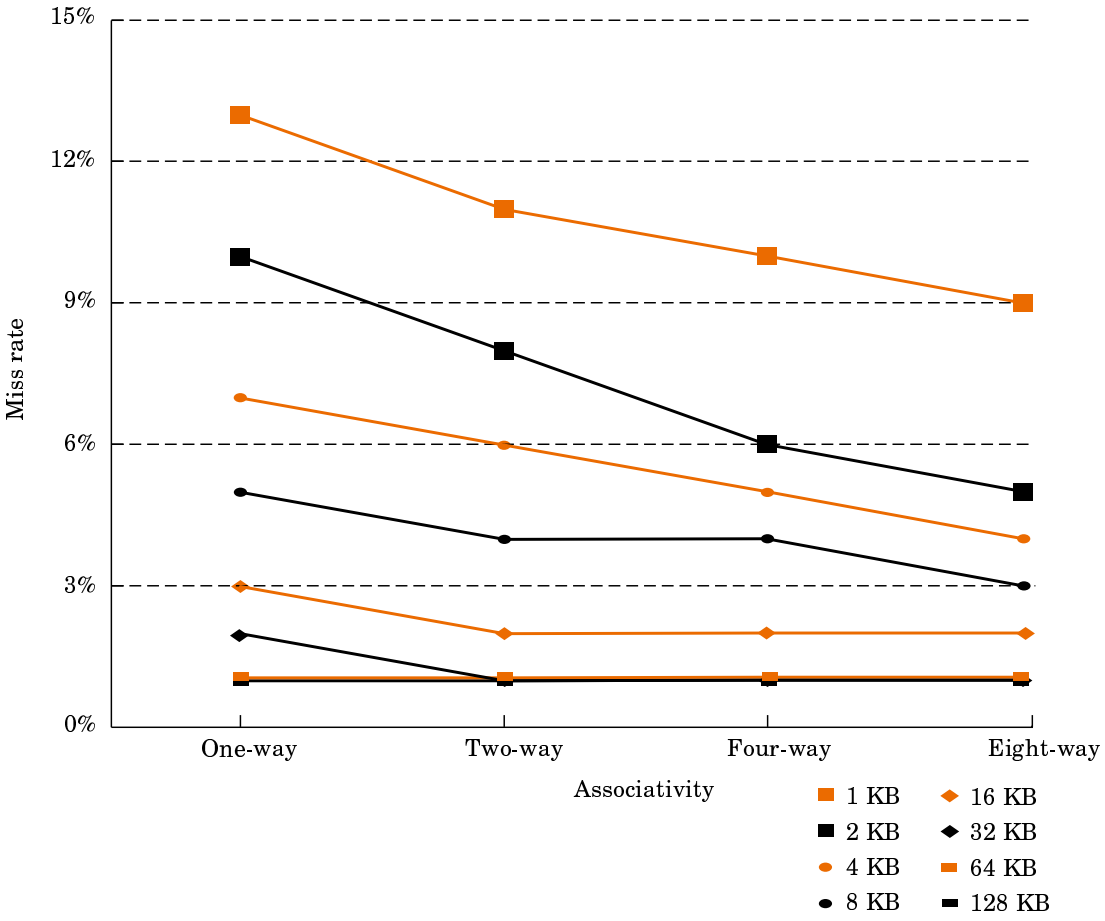
   ⊕ reduce traffic between cache and main memory; writes are faster

   ⊖ miss penalty higher since block must be written-back

3. **Cache Associativity**: How many lines in each set?

   The extremes are 1-way (i.e., direct mapped) and n lines (i.e., fully associative).

   Figure 7.29, pg. 604



$\Longrightarrow$ larger caches are better (no surprise)

$\Longrightarrow$ higher associativity better, with diminishing returns

4. **Split instruction and data caches**: fetch instructions from instruction cache; loads and stores go to data cache

   $\oplus$ can simultaneously (i.e., in parallel) fetch next instruction while completing a load or store

   $\ominus$ one of the two caches can be underutilized if code for loop is small, or working set is small

   - primary cache is often *split*
   - secondary cache is often *unified*

The Three C's of Cache Misses

1. **Compulsory miss**: caused by first access. Also known as cold-start or first-reference miss.
   - would still exist in a cache of infinite size

   Possible solution: prefetching
2. **Conflict misses**: different blocks map to the same line or set, resulting in frequent block replacements
   - impossible in a fully-associative cache

   Possible solution: increase associativity
3. **Capacity miss**: cache is too small to hold working set.
   - Even if there is locality of reference, if the working set is, say, 64 K and the cache is 32 K, there will be capacity misses.
   - possible in a fully-associative cache

   Possible solution: increase size of cache

Performance Gap "Tax"

Taken from Dr. Patterson's notes, Spring 1998.

Caches have no inherent value. They exist solely to try to reduce the processor-memory performance gap.

Processor	% area (≈ cost)	% transistors (≈ power)
Alpha 21164	37%	77%
StrongArm SA110	61%	94%
Pentium Pro (including L2 cache on second die)	64%	88%

As long as processors continue to be faster than memory, we will need caches and a memory hierarchy.

Hand and compiler optimizations to improve cache locality can have a **huge** impact on performance. Be aware!

Chp 4.8: Floating point arithmetic

- Notes cribbed from Dr. Gburzynski with additions.
- See the textbook, Chapter 4.8

Sometimes we call integer numbers in 2's complement notation "fixed-point" numbers.

The "decimal point" is located to the right of the least significant digit (and the fractional part is assumed to be zero).

```
      00000001  ->  00000001.  =  1.0
      11111110  ->  11111110.  = -2.0
since
      00000010  ->  00000010.  =  2.0
```

In some application we need REAL (fractional numbers) whose magnitude and range is not very large. Such numbers can be represented in fixed-point notation. i.e., partition the bits for before and after the fractional point.

For example,

$1.6875 = 1\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4}$

```
    00000001.1011  ->  1.6875  (a)
    11111110.0101  -> -1.6875  = ~(a) + 1
```

Thus,

```
    00000001.1000  ->  1.5
  + 11111110.0101  -> -1.6875
    -------------
    11111111.1101  -> -0.1875  (b)
```

since

```
    00000000.0011  ->  0.1875 = ~(b) + 1
```

$0.1875 = 0\times2^0+0\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4}$

The location of the point is irrelevant from the viewpoint of addition and subtraction.

For multiplication/division, the point must be shifted appropriately.

A typical application of fixed-point real numbers is in banking where people have to deal with dollars and cents.

The fixed-point notation has its limitations. In many scientific or engineering calculations one has to deal with very small and very large values, e.g., the size of the atomic nucleus (10^{-15}m) and the size of the visible universe (10^{25}m).

To represent all numbers from this range with a satisfying accuracy would require a large number of bits, most of which would be wasted (contain zeros) and there would only be a few significant digits:

```
0000···0000.0000···nnnnnnnn
nnnnnnnn···0000.0000···0000
```

Real numbers (whose magnitude can be very large or very small) are commonly represented in *floating-point* notation.

The representation of a number consists of two *integer* (fixed-point) parts or fields:

MANTISSA specifying the significant digits of the number in a position-less fashion;

EXPONENT specifying the location of the point, i.e., the magnitude of the number.

The number reads as:

$$\boxed{b^e \times m}$$

where b is some fixed base (typically a power of 2).

Different flavours of the generic floating-point notation (as introduced above) are characterised by the following parameters:

- the number of bits used to represent the mantissa;
- the number of bits used to represent the exponent;
- the base b;
- the representation of the mantissa (2's complement, sign-magnitude, the location of the point);
- the representation of the exponent.

Most issues related to floating point numbers can be illustrated assuming that both the exponent and mantissa are decimal numbers and that $b = 10$.

Below we have a few sample numbers:

$$10^{12} \times 1$$
$$10^{0} \times 1995$$
$$10^{-3} \times -22384 \ (= -22.384)$$

All the above representations assume that the (decimal) point of the mantissa is located after the rightmost digit. Of course, this doesn't have to be the case, e.g.:

$$10^{13} \times .1$$
$$10^{4} \times .1995$$
$$10^{2} \times -.22384$$

or

$$10^{10} \times 10.00$$
$$10^{2} \times 19.95$$
$$10^{0} \times -22.384$$

Notably, unless we impose additional rules, floating-point numbers don't have unique representations, e.g.:

$$10^0 \times 1995$$
$$10^{-1} \times 19950$$
$$10^{-10} \times 19950000000000$$

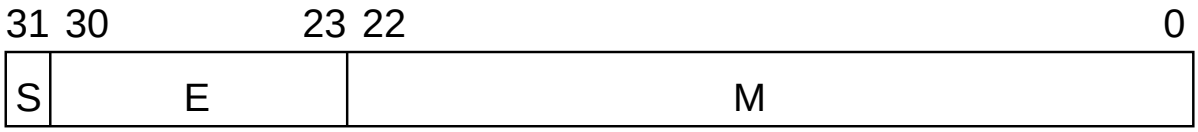
all denote the same number.

We say that a floating point number is NORMALISED, if the most significant digit of its mantissa is nonzero.

Of course, on a computer we have some definite number of (binary) positions for representing the mantissa, so the above definition makes sense.

Zero has to be treated in a special way.

IEEE Floating Point Format (32-bit):



A nonzero number is computed as:

$$\boxed{-1^{\mathsf{S}} \times 2^{\mathsf{E}-127} \times 1.\mathsf{M}}$$

where E is an unsigned integer number.

The mantissa is always normalised with the (binary) point located after the first digit (thus its value is between 1 (inclusively) and 2 (exclusively).

As the most significant bit of the mantissa is always 1, there is no need to represent it; thus, it doesn't appear in M.

The 127 is the *exponent bias*. Two values of exponent E are reserved: 0 – to represent the number zero, and 255 – to represent *infinity*.

Recall,

$$\boxed{-1^{\mathsf{S}} \times 2^{\mathsf{E}-127} \times 1.\mathsf{M}}$$

Let us have a look at a few simple numbers:

`0 01111111 00000000000000000000000` = 1.0

$$= -1^0 \times 2^{127-127} \times 1.0$$

$$= 1 \times 2^0 \times 1.0 = 1.0$$

`1 01111111 00000000000000000000000` = -1.0

$$= -1^1 \times 2^{127-127} \times 1.0$$

$$= -1 \times 2^0 \times 1.0 = -1.0$$

`0 10000000 00000000000000000000000` = 2.0

$$= -1^0 \times 2^{128-127} \times 1.0$$

$$= 1 \times 2^1 \times 1.0 = 2.0$$

```
* 00000000 00000000000000000000000 = +/- 0
* 11111111 00000000000000000000000 = +/-inf
* 11111111 ...any non 0 pattern... = NaN
```

How to represent 1995.5?

In fixed-point notation the number looks like:

```
11111001011.1
```

When we turn it into `1.11110010111`, we have to multiply it by 2^{10} to get the original value.

By simply moving the decimal point, it is easy to see that the number will be represented as:

```
0 10001001 11110010111000000000000
```

The hard (but instructive way) is to use the equation:

$$\boxed{-1^{\mathsf{S}} \times 2^{\mathsf{E}-127} \times 1.\mathsf{M}}$$

$$= -1^0 \times 2^{137-127} \times 1.948730468...$$

$$= 1 \times 2^{10} \times 1.948730468... = 1995.5$$

Floating point extremes

What is the **largest** number (in decimal) that can be represented in 32-bit IEEE floating point?

`0 11111110 11111111111111111111111`

$$= -1^0 \times 2^{254-127} \times 1.11111111111111111111111_2$$

$$= 1 \times 2^{127} \times ((2^{24} - 1) \times 2^{-23})$$

$$= 2^{127-23} \times (2^{24} - 1)$$

$$= 2^{104} \times (2^{24} - 1)$$

$$= 3.402823466 \times 10^{38}$$

The exponent can be a maximum of 254. It cannot be 255 (`11111111`) because that is reserved to represent `+/-inf` and `NaN`.

Note that a number with n 1's in the binary form is equal to $2^n - 1$ (e.g., $2^8 - 1 =$ `11111111`). Thus, $2^{24} - 1$ is a binary number with 24 ones.

Lastly, the 2^{-23} simply corrects the fractional point from the mantissa (i.e., 23 bits)

What is the **smallest positive** number (in decimal) that can be represented in 32-bit IEEE floating point?

`0 00000001 00000000000000000000000`

$$= -1^0 \times 2^{1-127} \times 1.0$$

$$= 2^{-126} \times 1.0$$

$$= 1.175494351 \times 10^{-38}$$

The exponent can be a minimum of 1. It cannot be 0 because that is reserved to represent `+/-0` and denormalized numbers.

So, between $1.175494351 \times 10^{-38}$ and $3.402823466 \times 10^{38}$ we can store a **range** from the size of the atomic nucleus (10^{-15}m) to the size of the visible universe (10^{25}m).

But, numbers that cannot be represented exactly typically have only on 6 or so significant digits. This has serious implications.

Cautionary tale

Clearly, since real numbers (in mathematics) can have an infinite number digits in the fractional part, it is impossible to represent all real numbers using a finite number of bits.

With a finite number of bits, some numbers can be represented exactly (e.g. 1.5) and other numbers can only be approximated.

This is true even if they fall into the **range** of possible values for a floating point format.

Typically, for 32-bit IEEE the smallest detectable difference between two floating point numbers is (about) 3×10^{-8}. This is called the *machine accuracy*.

Consider:

```
#include <stdio.h>
int main( int argc, char ** argv )
{
 float a = ( 1 + 3E-9 ); float b = ( 1 + 3E-8 );
 float c = ( 1 + 3E-7 ); float d = b + 0.0005;
 float e = ( 1 + 3E-5 ); float f = 1.000322;

 printf( "a=%g, b=%g, c=%g, d=%g, e=%g, f=%g\n",
     a, b, c, d, e, f );
 if( a == b ) printf( "a == b\n" );
         else printf( "a != b\n" );
 if( b == c ) printf( "b == c\n" );
         else printf( "b != c\n" );
 if( f == 1.000322 ) printf( "f correct\n" );
         else printf( "f incorrect\n" );
 return( 0 );
}
% gcc main.c; a.out    (Pentium MMX, Linux 2.0)
a=1, b=1, c=1, d=1.0005, e=1.00003, f=1.00032
a == b
b != c
f incorrect
```

- `a == b` $\Longrightarrow$ that $1 + 3 \times 10^{-9}$ is indistinguishable from $1 + 3 \times 10^{-8}$
- `b != c` but `b = 1, c = 1` $\Longrightarrow$ that `b` and `c` are different, but output shows them to be the same
- `f incorrect` $\Longrightarrow$ that either `1.00032` cannot be exactly represented, or there are some compiler, or other issues here. Or the "obvious" behaviour is not correct. Frankly, I'm not sure, but I'm troubled!

Lessons? Understand the limitations...

1. of computer floating point and the specific format implemented by your CPU (e.g., IEEE? mostly IEEE?)
2. of your particular compiler and libraries
3. of computations using floating point (i.e., round-off error)

All of these things and more are part of the field of numerical analysis and scientific computation.

There is a popular series of books called "Numerical Recipes", with editions for C, Fortran, etc.

Read `http://www.colorado.edu/ITS/docs/scientific/fortran/numrec.html` to see why you should use that resource with caution.

Back to the program...

The IEEE standard defines two higher-precision formats:

DOUBLE (64 bits):

Exponent size = 11 bits (exponent bias = 1023)

Mantissa size = 52 bits

QUAD (128 bits):

Exponent size = 15 bits (exponent bias = 16383)

Mantissa size = 112 bits

In both cases, the interpretation of *exponent* and *mantissa* is the same as for the 16-bit format (but the ranges are different).

MIPS Floating Point

The MIPS architecture supports floating point arithmetic through co-processor 1 or FPU, which usually on the same die/chip as the CPU.

1. 32 single-precision floating-point registers: `$f0` – `$f31`. The registers can be paired for double-precision numbers.
2. Special loads and stores: `lwc1`, `swc1`
3. Special arithmetic instructions:
   `add.s`, `sub.s`, `mul.s` (single-precision),
   `add.d`, `sub.d`, `mul.d` (double-precision)

For example (from page 288):

```
                float x, y, z;
                z = x + y;

lwc1    $f4,x($sp)   # Load 32-bit FP into $f4
lwc1    $f6,y($sp)   # Load 32-bit FP into $f4
add.s   $f2,$f4,$f6  # $f2 = $f4 + $f6, single precision
swc1    $f2,z($sp)   # Store 32-bit FP from $f2
```