Chapter 2:

Basic Computer Operations and Assembly Language Programming

Topics:

Components of a computer
Programmer's view of computer system
Simple Program Translation with MIPS
- **Allocating variables to registers**
- **Arithmetic statements**
- **If, if-else**
- **For, while, do loops**
- **Input/output**

Reading: Patterson and Hennessy
- **1.1 – 1.5**
- **2.1 – 2.3, 2.6 – 2.7**
- **Appendix A.9**

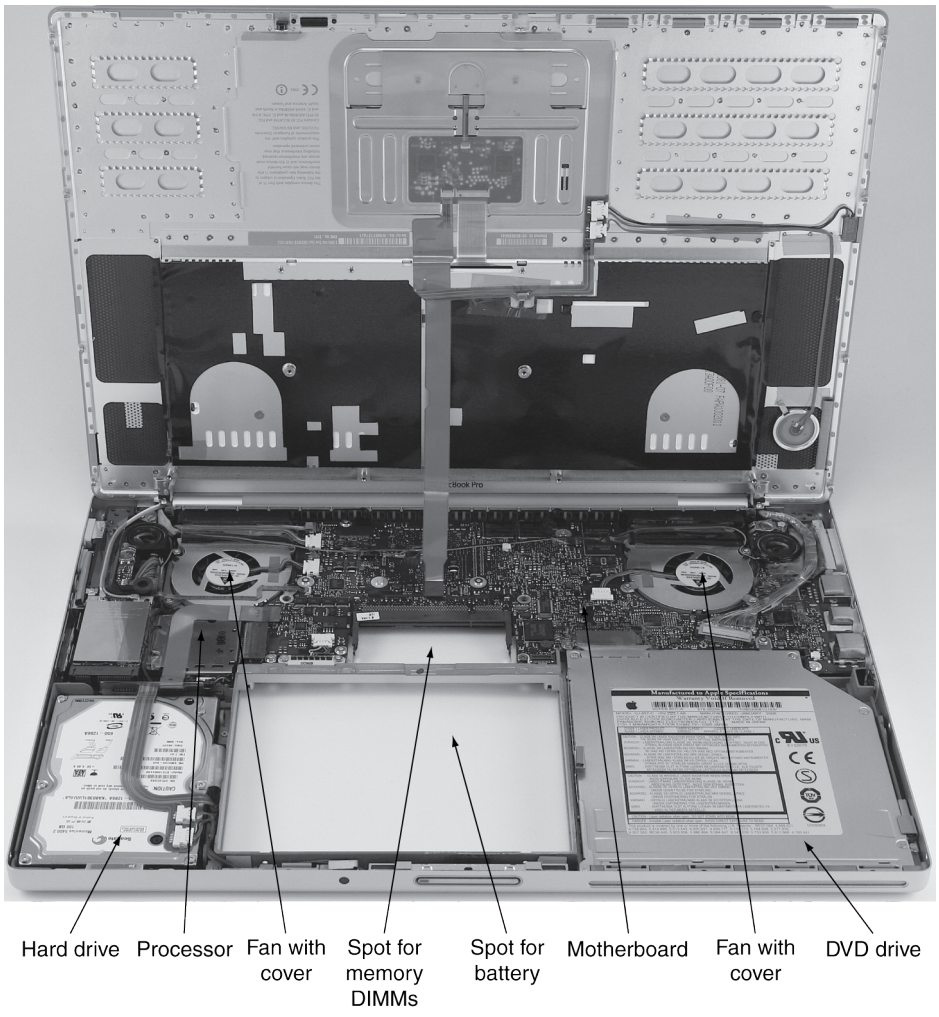
Components of a Computer

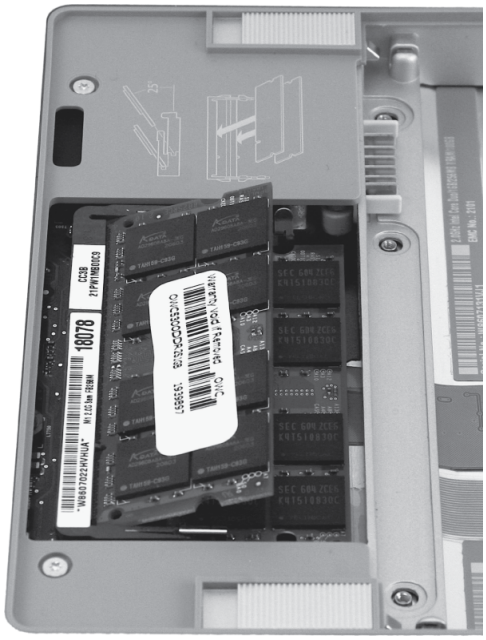
Types of computers:

- server
- desktop / laptop
- embedded system

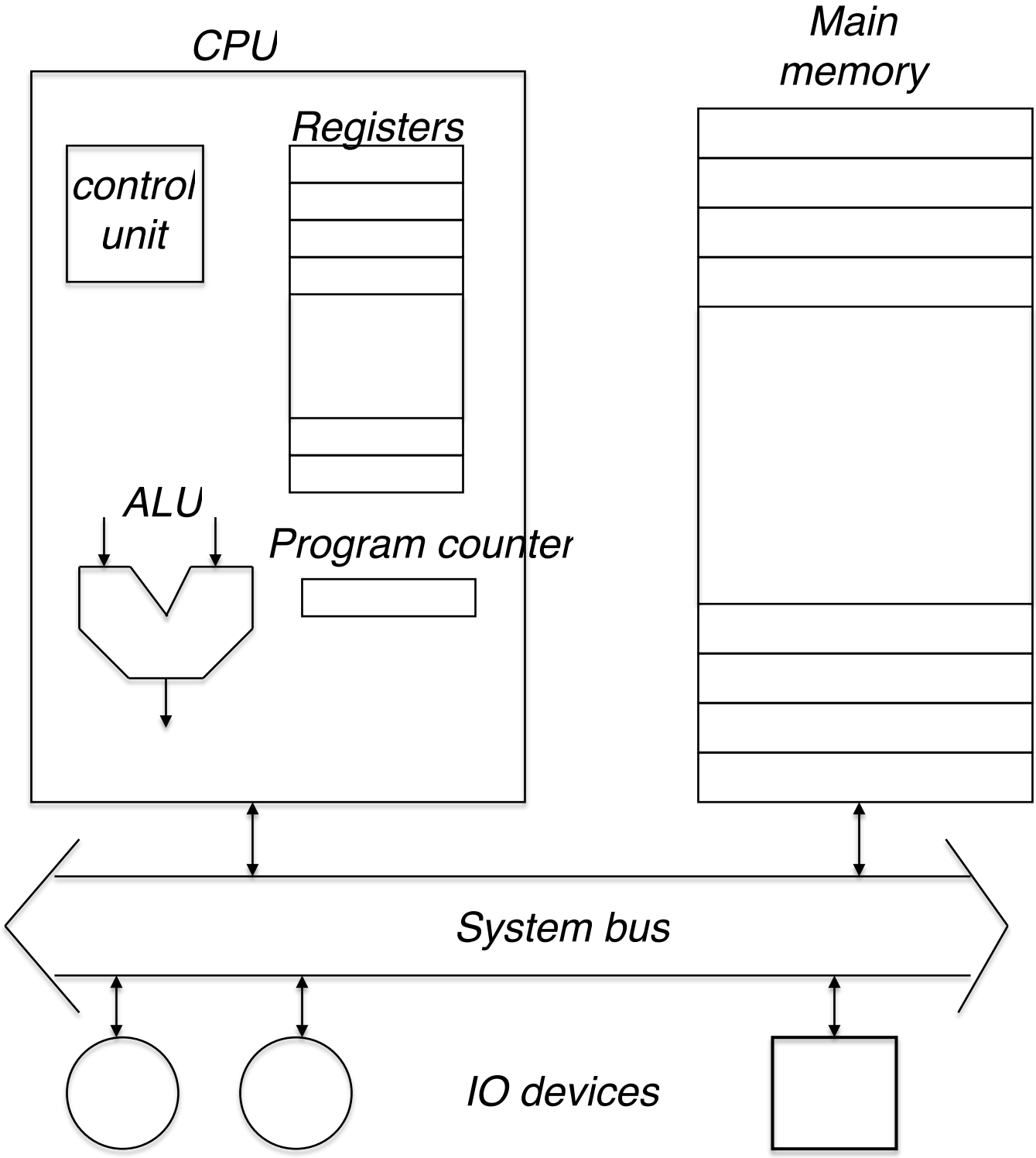
All have similar components:

- Central processing unit (CPU)
- Memory and other storage
- Input/output devices (display, keyboard, mouse)


Hard drive
Processor
Fan with cover
Spot for memory DIMMs
Spot for battery
Motherboard
Fan with cover
DVD drive



Programmer's View of Computer System



System bus:

shared set of wires that connect most computer components

Input/output (IO) devices:

devices that enable communication with outside world

hard drive, CD/DVD drive, USB drives

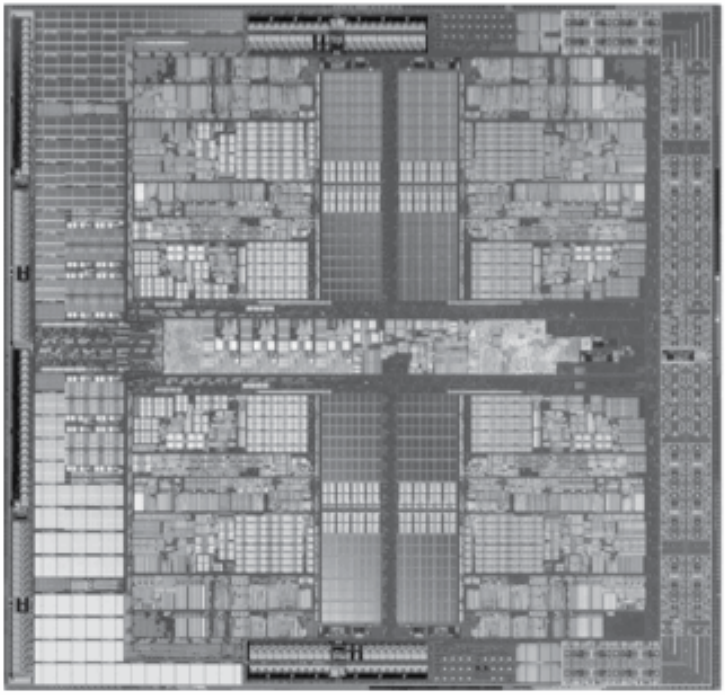
mouse, keyboard, touchpad, monitor, touchscreen, network adapters

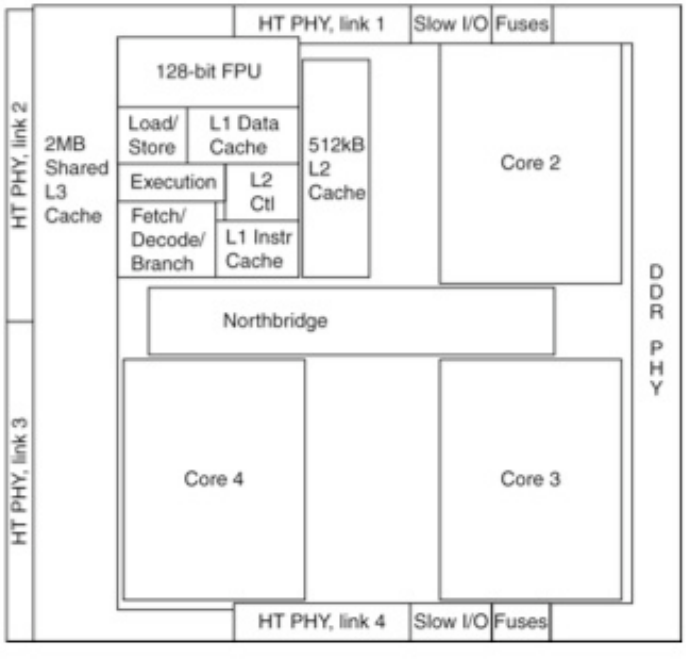
Central Processing Unit (CPU):

“brain” or main active component in the system

Some CPUs: Sun Sparc, PowerPC, Intel x86
For CSc 256, use MIPS CPU (Silicon Graphics)

A Recent Processor: AMD Barcelona (4 cores!)





Main memory:

main storage for actively running programs

Units of storage:

bit
nibble *4 bits*
byte *8 bits*

halfword *16 bits*
word *32 bits*
doubleword *64 bits*

MIPS CPU:
* each memory address is 32 bits

Memory is *byte-addressed*:
* each increment of the address refers to the next byte

How memory hardware works:

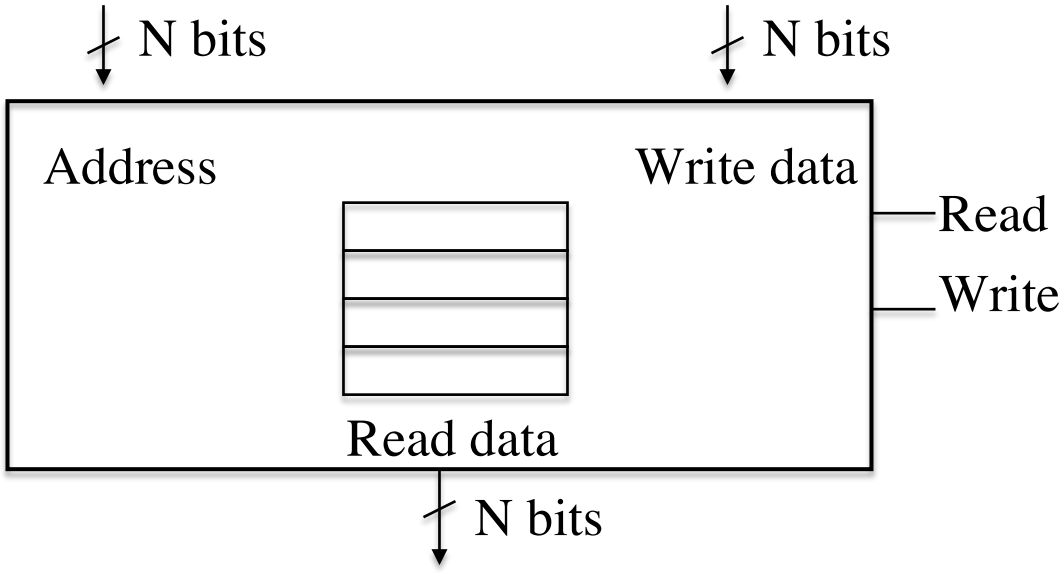
Memory is like an array of variables.

First we have to *choose* a location.

Array: specify the *index* (ex. x[i])

Memory: specify the *memory address*

(MEM[ADDR])



Then we can read/write the location.

Array read: … = x[i];
Array write: x[i] = … ;

Memory read:
Place address bits on address pins
Set Read control to 1
Data appears on Data Read pins

Memory write:
Place address bits on address pins
Place data on Data Write pins
Set Write control to 1
Data gets written to location

Memory viewed as array of bytes:

address	*8 bits* contents
0x0000 0000	*0x13*
0x0000 0001	*0x24*
0x0000 0002	*0x57*
0x0000 0003	*0x68*
0x0000 0004	*0x9a*
0x0000 0005	*0xbb*
0x0000 0006	*0xcc*
0x0000 0007	*0xef*
0x0000 0008	
0x0000 0009	
0x0000 000a	
0x0000 000b	

Memory viewed as array of 32-bit words:

address	**contents** *32 bits (4 bytes)*
0x0000 0000	*0x1324 5768*
0x0000 0004	*0x9abb cce1*
0x0000 0008	
0x0000 000c	
0x0000 0010	
Etc	
Etc	
0xffff fffc	

Memory byte-order:

MIPS memory is *big-endian*:
the byte with the lowest address is at the most significant end (left-most, end with highest value)

[most commercial CPUs are big-endian, except Intel x86; little-endian:]

Contents of a byte at address 0x0000 0004:

0x9a

Contents of a byte at address 0x0000 0006:

0xcc

Contents of a word at address 0x0000 0004 (contents of 4 bytes at 0x0000 0004 to 0x0000 0007): *0x9abb ccef*

Aligned versus misaligned:

If a word in memory is aligned, the 4 bytes within the word do not cross word boundaries (words at 0, 4, 8, 0xc, 0x10, 0x14...)

If a word is misaligned, its 4 bytes cross word boundaries. (words at 1, 2, 3, 5, 6, 7 ...)

How to tell if a word address is aligned?

1) look at least significant (rightmost) digit

0, 4, 8, c

2) look at least significant 2 bits

= 00

Size of memory address (32 bits in MIPS, Pentiums) determines the amount of memory that can be addressed by programs running on a CPU.

32-bit address means a program can "see"

4GB of memory

Note, for memory:
1 Kilobyte (KB) = *2^10 = 1024*
1 Megabyte (MB) = *2^20 = 1024 * 1024*
1 Gigabyte (GB) = *2^30*
1 Terabyte (TB) = *2^40*

See:
http://en.wikipedia.org/wiki/Terabyte

Note the difference between *Standard SI (International System of Standards)* and *Binary usage*.

Standard SI is used for hard drives and other storage devices, or data transfer rates.
Binary usage is for memory.

Many computers today have 64-bit memory addresses.

However, this doesn't mean they all have ~~4~~ 16 Exabytes (~~4~~ 16 * 2^60 bytes) of physical memory I the hardware!

Later in the semester, we'll talk about *virtual memory*, which allows applications to address more memory than is physically available.

Parts of CPU

Control unit:

main active component in CPU

Arithmetic Logic Unit (ALU):

like a calculator

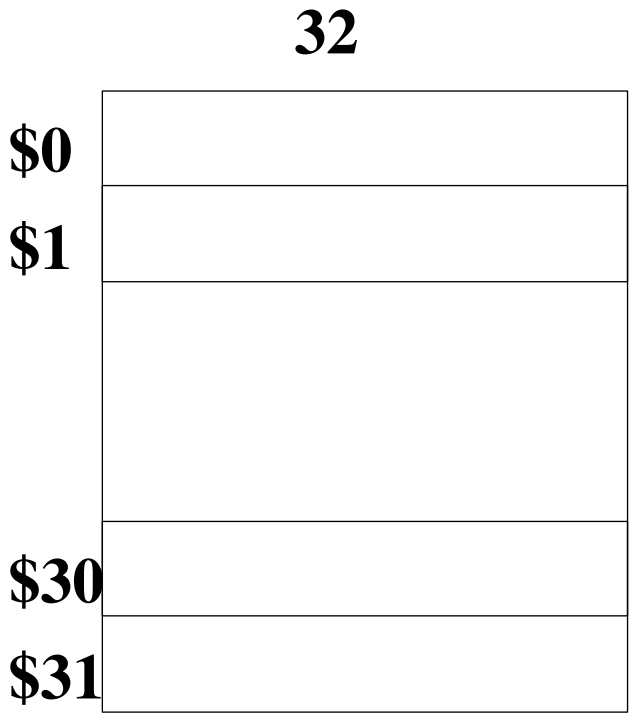
Registers:

small fast storage inside CPU

Program counter:

special register that contains address of the current instruction

MIPS has 32 registers, each a 32-bit word



Two registers with restricted use:

$0 *== 0*

$1 *reserved for assembler*

Registers have alternate names:

$2 - $3 also called $v0 - $v1
$4 - $7 also called $a0 - $a3
$8 - $15 also called $t0 - $t7
$16 - $23 also called $s0 - $s7
$24 - $25 also called $t8 - $t9
$26 - $27 also called $k0 - $k1
$28 also called $gp
$29 also called $sp
$30 also called $s8
$31 also called $ra

For now, use $s0 - $s8 (always safe), or $t0 - $t9.

More on these later, when we talk about function calls.

High Level Language Program:

statement 1
statement 2
statement 3

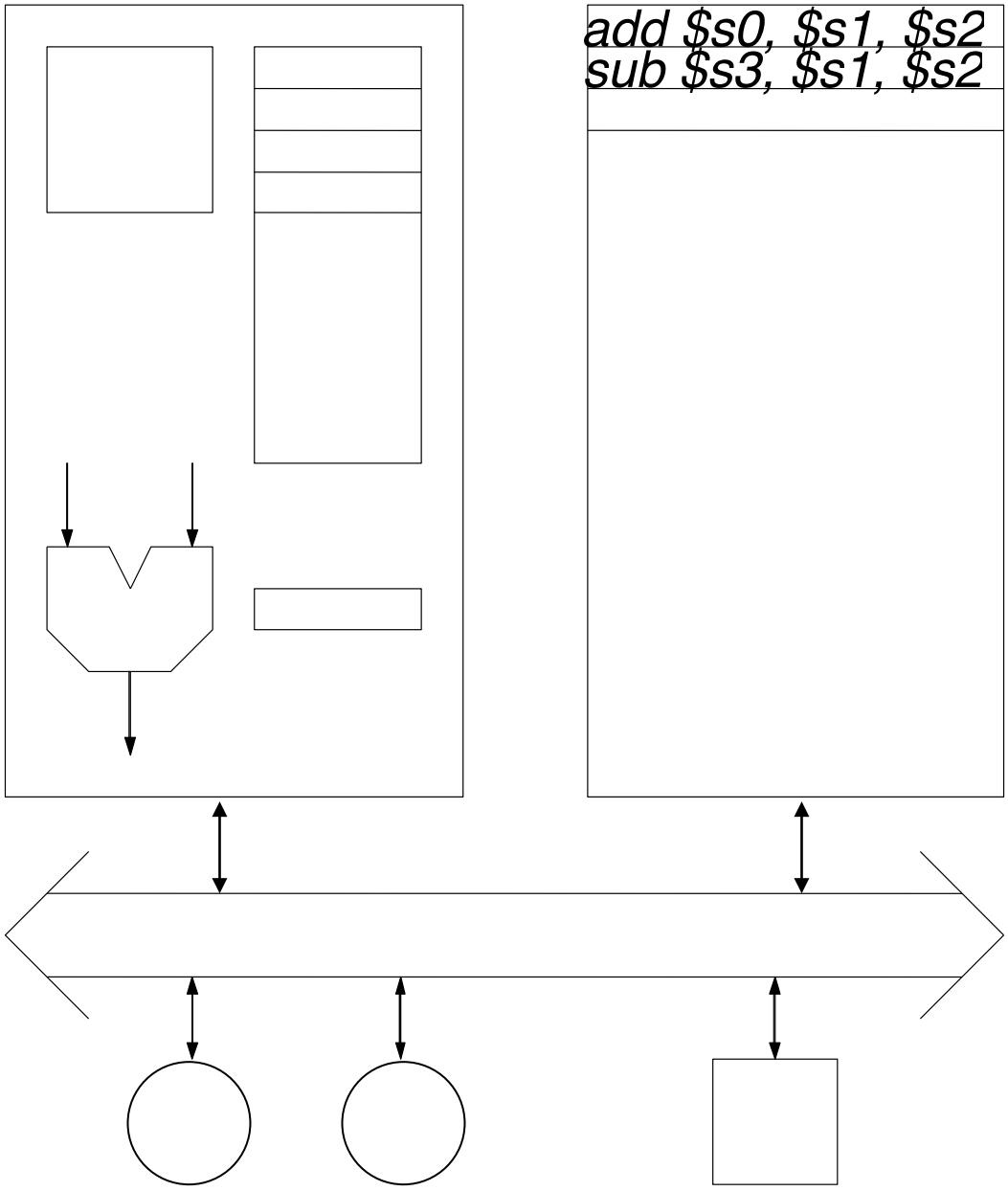
Assembly Language Program:

instruction 1
instruction 2
instruction 3

Von Neumann (or Eckert/Mauchly) architecture:

1) programs and data are stored in the same memory
2) instructions are executed in sequential order, except for branches/jumps

What happens when a user runs a program?



1) user types `a.out\n`
2) CPU "finds" a.out file
3) CPU loads memory with program code and data from file

Typical assembly language format:

operation operand1 operand2 operand3 ...

Operation: what the instruction does (+, -)
Operands: variables

Variables are allocated to registers, if possible. (Can also allocate variables to memory, but *slower*.)

For example ($s0-$s3 contain int variables):

add $s0,$s1,$s2 means $s0 = $s1 + $s2
sub $s3,$s1,$s2 means $s3 = $s1 - $s2

Dealing with constants:

addi $s0, $s1, 1 means $s0 = $s1 + 1

Note: no *subi* instruction! Why?

Introduction to program translation

MIPS (like PowerPC, Sun Sparc, Intel IA64) is a *load/store* architecture. This means:

arithmetic and logic instructions work with registers/constants only.
load/store instructions move data between memory and registers.

A good compiler will try to allocate frequently used variables to registers.

Advantages: fast access, less code
Disadvantages:
a) may not have enough registers for all vars
b) must keep track of where each var is
(ex. $s0 is x, $s1 is y etc)

stopped 9/1/2016

First MIPS Program (variables in registers)

```
int v=0,w=1,x=2,y=3,z=4;

int main(void)
{
  w = z;
  v = w + x;
  y = w - x;
  z = z - 1;
}
```

Compiler decides:

v	$s0
w	$s1
x	$s2
y	$s3
z	$s4

Note: comment delimiter is “#”, similar to “//” in C++/Java

[see Example 2.1:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.1]

```
        .data
[left blank for this example...]
        .text
main:  addi  $s0, $0, 0   # v = 0;
                          # w = 1;
                          # x = 2;
                          # y = 3;
                          # z = 4;

                          # w = z;
                          # v = w + x;
                          # y = w - x;
                          # z = z - 1;

       addi  $v0, $0, 10
       syscall            # program ends
```

Def: MIPS arithmetic/logic instructions

int x,y,z;

Compiler decides:

x	$s0
y	$s1
z	$s2

MIPS: add $s0,$s1,$s2
[all 3 operands must be registers]
contents of $s0
= contents of $s1 + contents of $s2
C/C++: x = y+z

MIPS: sub $s0,$s1,$s2
[all 3 operands must be registers]
contents of $s0
= contents of $s1 - contents of $s2
C/C++: x = y-z

MIPS: addi $s0, $s1, 1
[operand1 and operand2 are registers, operand3 is any integer constant]
contents of $s0 = contents of $s1 + 1
C/C++: x = y+1

MIPS: mul $s0,$s1,$s2
contents of $s0
= contents of $s1* contents of $s2
*C/C++: x = y*z*

MIPS: div $s0,$s1,$s2 [integer division]
contents of $s0
= contents of $s1/ contents of $s2
C/C++: x = y/z

MIPS: rem $s0,$s1,$s2
contents of $s0
= contents of $s1% contents of $s2
C/C++: x = y % z

MIPS: li $s0,4
[operand1 is any register, operand2 is any constant]
contents of $s0 = 4
C/C++: x = 4

MIPS: move $s0,$s1
[operand1 and operand2 are any registers]
contents of $s0 = contents of $s1
C/C++: x = y

[*div*, *rem*, *li*, *move* are *pseudoinstructions*, like *macros*. More later…]

Translating compound arithmetic statements

Example:
D = b * b + 4 * a * c ;

Use temporary variables (usually $t? registers), break into simple statements:

temp0 = a * c;
temp0 = 4 * temp0;
temp1 = b * b;
D = temp1 + temp0;

Compiler decides:

a $s0
b $s1
c $s2
D $s3

```
# temp0=a*c;
# temp0=4*temp0;
# temp1=b*b;
# D=temp1+temp0;
```

MIPS Input/output

To perform input/output (I/O), the *syscall* instruction is used to call system routines.

Before syscall is executed,
1) $v0 contains a code number to indicate the type of operation (read an int, print an int etc)
2) $a0 contains additional arguments
(e.g. integer to be printed)
3) result from user input appear in $v0

C++:

```
int n=9; // n is assigned to $s0

cout << n; // System.out.print(n)
```

MIPS:

```
li    $v0, 1       # cout << n
move  $a0, $s0
syscall
```

C++:

```
cin >> n;    // Scanner input = new
             //    Scanner(System.in);
             // n = input.nextInt();
```

MIPS: (code for reading int is 5, n is $s0)

```
li    $v0, 5       # cin >> n;
syscall
move  $s0, $v0
```

C++:

```
char str[] = "abc";

cout << str; // System.out.print(str);
```

MIPS:

```
      .data
str:  .asciiz "abc"      # char str[]
                         #     = “abc”;

      li        $v0, 4   # cout << str;
      la        $a0, str
      syscall
```

Selection (if, if-else)

Simple if statement in C++:

```
if (x != y)
  sum = sum + x;
```

Suppose x is $s0, y is $s1, sum is $s2.

We need a MIPS *conditional branch*. It does different things depending on a *test condition*.

In general, a MIPS conditional branch looks something like:

if (condition true) goto label;
else execute next instruction

beq op1, op2, label [branch if equal]

if (contents of op1 == contents of op2) goto label
else execute next instruction

op1, *op2* are any registers
label marks a location in memory that contains an instruction, ex.:

```
main: add   $s0, $s1, $s2
```

Example: beq $s0, $s1, label
(if $s0 == $s1 goto label; else goto next instruction)

Branch if not equal: bne op1,op2, label

if (contents of op1 != contents of op2)
    goto label;
else execute next instruction

Original:

```
  if (x != y)
    sum = sum + x;
```

Rewrite:

```
      if (x == y) goto skip;
      sum = sum + x;
skip: [next statement]
```

Compiler decides: x is $s0, y is $s1, sum is $s2

MIPS:

```
       beq $s0,$s1, skip
       add $s2, $s2, $s0
skip:  [next instruction]
```

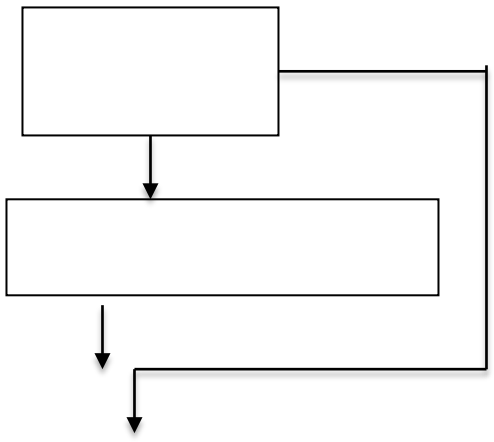
[see Example 2.2:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.2.s]

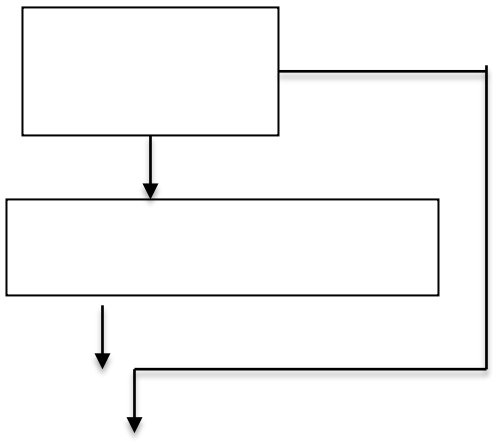
Making structured flowcharts:

1) each box is either an arithmetic statement or a test
2) arithmetic statements have only one exit
3) tests have two exits, yes and no
4) boxes should be arranged in a straight line
5) no conditions are in main line of flow
6) yes conditions are side branches

To translate from C++ code to flowchart (or MIPS):
1) write down each statement or test, in order.
2) change test conditions if necessary

Flow chart for `if (x!=y) sum = sum+x;`





Unconditional branches (goto):

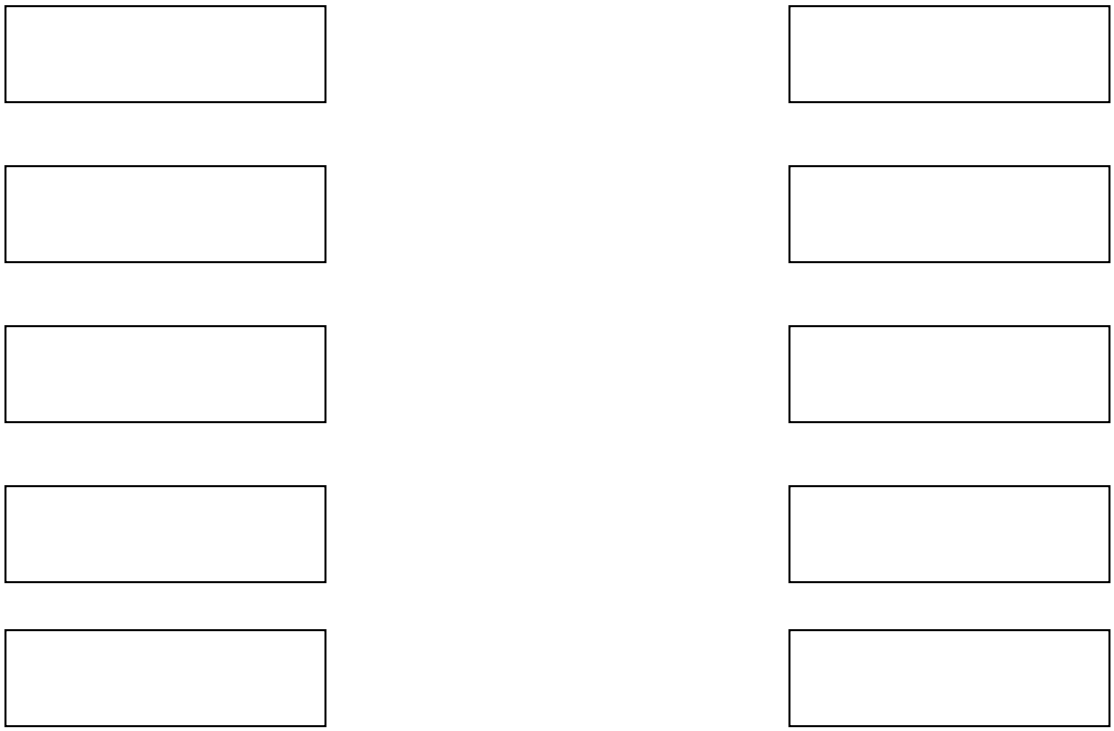
j label [jump; goto label]

Translating if-else statements:

Example (suppose num is $s0, flag is $s1, max is $s2)

```
if (num == max)
  flag = 1;
else if (num != 0)
  flag = -1;
else flag = 0;
```

Flow charts:



MIPS version (num is $s0, flag is $s1):

```
# if (num==max)
#   flag=1;

# else if (num!=0)
#   flag=-1;

# else flag=0;
```

[see Example 2.3:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.3.s]

With constants (use temp!):

```
                        # if (x != -1)

                        #   flag = 1;
```

Other conditions:

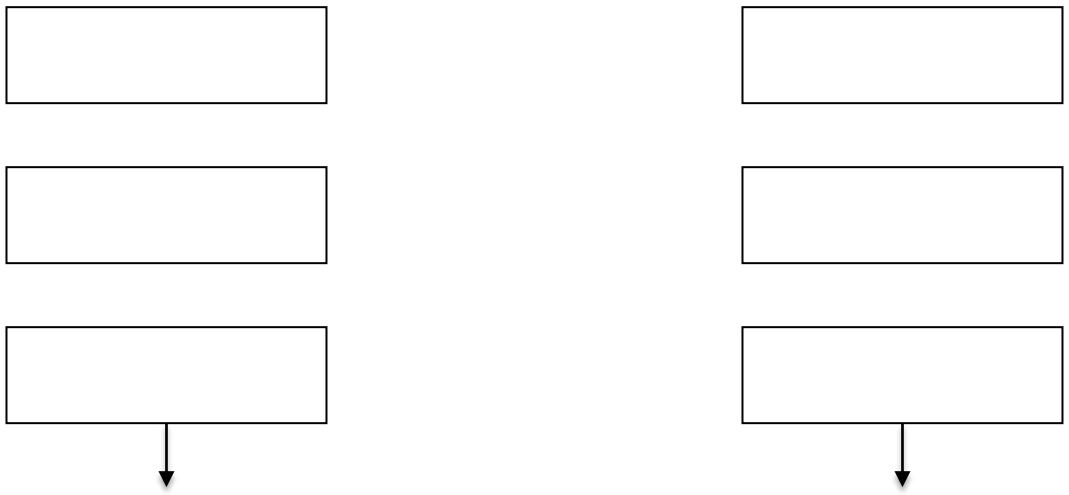
```
blt  $s0, $s1, label # if ($s0<$s1)
bgt  $s0, $s1, label # if ($s0>$s1)
ble  $s0, $s1, label # if ($s0<=$s1)
bge  $s0, $s1, label # if ($s0>=$s1)
```

[These are actually pseudoinstructions…]

If statements with “and” conditions:

```
if ((x > y) && (x < 0))
  flag = 23;
```

(Suppose x is $s0, y is $s1, flag is $s2)
Flow chart:

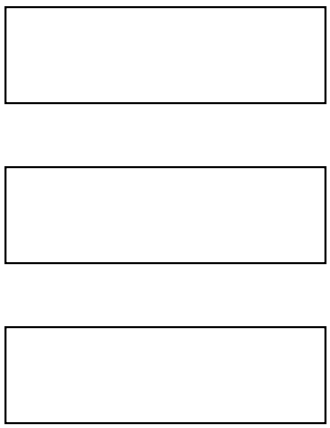


MIPS code:

If statements with “or” conditions:

```
if ((a != 0) || (b < 5))
  grade = -1;
```

(Suppose a is $s0, b is $s1, grade is $s2)
Flow chart:



MIPS code:

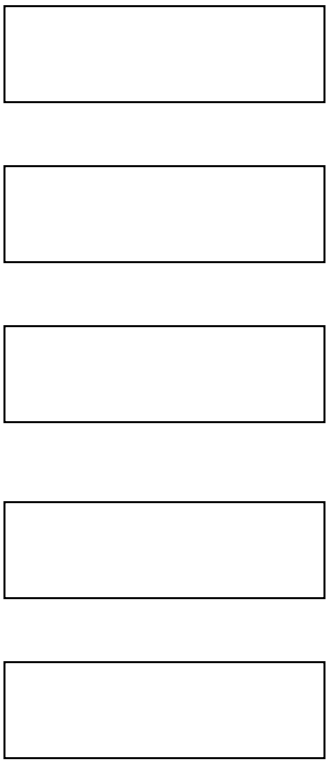
Repetition: loops

C++:

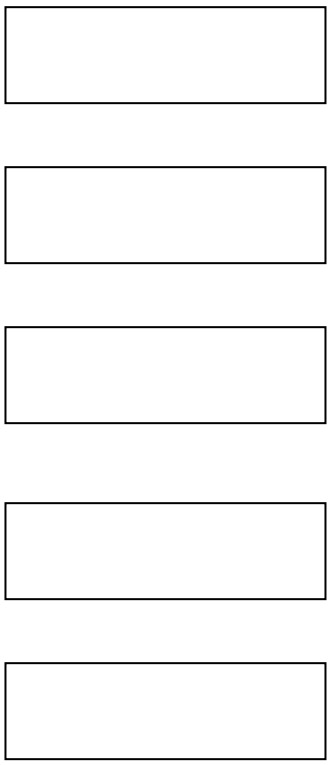
```
sum = 0;
for (i=1; i<= limit; i++)
  sum = sum + i;
```

Suppose sum is $s0, i is $s1, limit is $s2.

Flow chart and MIPS code (Version 1):



Flow chart and MIPS code (Version 2):



Version 2 is more efficient. All your loops should follow version 2, with test condition at the bottom of the loop.
[see Example 2.4:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.4.s]

do-while loop:

```
do {
        /* actions */
} while (sum < 100);
```

while loop:

```
while (x > 0) {
        /* actions */
}
```

[see Example 2.5: http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.5.s]

Truth table for simple logical operations

y	z	y and z	y or z	y nand z	y nor z	y xor z
0	0					
0	1					
1	0					
1	1					

Bitwise logical operations:
logical operation applied to each bit of operand
(Suppose x is $s0, y is $s1, z is $s2)

C/C++: x = ~y (different from x = !y)
MIPS: not $s0,$s1
contents of $s0 = not (contents of $s1)

Example:
int x, y = 0x12345678;

x = ~y;

x becomes:

C/C++: x = y& z
MIPS: and $s0,$s1,$s2
contents of $s0
= contents of $s1 and contents of $s2

C/C++: x = y | z
MIPS: or $s0,$s1,$s2
contents of $s0
= contents of $s1 or contents of $s2

C/C++: no nor operator
MIPS: nor $s0,$s1,$s2
contents of $s0
= contents of $s1 nor contents of $s2

C/C++: x = y ^ z
MIPS: xor $s0,$s1,$s2
contents of $s0
= contents of $s1 xor contents of $s2

Shifts

C/C++: x = y << AMT (x in $s0, y in $s1)
MIPS shift left logical: sll $s0, $s1, AMT
contents of x = contents of y shifted left by AMT bits, "empty" bits filled with zeros

Example:
`y = 0x89abcdef;` // in $s1

After sll $s0,$s1,3
$s0 becomes:

1000	1001	1010	1011	1100	1101	1110	1111

C/C++:
unsigned int x,y;
x = y >> AMT; // x,y declared unsigned

MIPS shift right logical: srl $s0,$s1, AMT
contents of x= contents of y shifted right by AMT bits, "empty" bits filled with zeros

Example:
y = 0x89abcdef // in $s1

After srl $s0,$s1,3
$s0 becomes:

1000	1001	1010	1011	1100	1101	1110	1111

C/C++:
int x,y;
x = y >> AMT; // x,y declared int

MIPS shift right arithmetic: sra $s0,$s1, AMT
contents of x = contents of y shifted right by AMT bits, "empty" bits filled with sign bit

Example:
y = 0x89abcdef // in $s1

After sra $s0,$s1,3,
$s0 becomes:

1000	1001	1010	1011	1100	1101	1110	1111

Relationship between shifts and arithmetic operations:

shift left n bits equivalent to

shift right n bits equivalent to

must watch for overflow!

[Example 2.6 (C and MIPS versions):
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.6.cpp
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/2.6.s]

List of MIPS instructions: Appendix A 51-80

Arithmetic/logic instructions (p. A-51 to A-55)

[op1, op2, op3 are operands, usually registers or constants]

```
li    op1, CONSTANT      # pseudo
move  op1, op2           # pseudo
add   op1, op2, op3
sub   op1, op2, op3
addi  op1, op2, CONSTANT
mul   op1, op2, op3
div   op1, op2, op3      # pseudo
rem   op1, op2, op3      # pseudo
not   op1, op2
and   op1, op2, op3
or    op1, op2, op3
nand  op1, op2, op3
nor   op1, op2, op3
xor   op1, op2, op3
sll   op1, op2, AMT
srl   op1, op2, AMT
sra   op1, op2, AMT
```

Branch instructions (p. A-59 to A-63)

[op1, op2, op3 are operands, usually registers or constants]

```
j      label
beq    op1, op2, label
bne    op1, op2, label
bgt    op1, op2, label
bge    op1, op2, label
blt    op1, op2, label
ble    op1, op2, label
```

Instruction for requesting system services:

syscall

Some useful Unix commands

ls list files in current directory
cat <filename> display contents of file called <filename> on screen
more <filename> like "cat", but page by page
cd <dir-name> change to directory with name <dir-name>
cp <oldfile> <newfile>
make a copy of <oldfile>; the new copy is called <newfile>
mv <oldfile> <newfile>
change the name of <oldfile> to <newfile>

Some useful spim commands

exit -- Exit from the simulator
read "FILE" -- Read FILE of assembly code into memory
load "FILE" -- Same as read
run <ADDR> -- Start the program at optional ADDRESS
step -- execute the next TAL instruction in the program
step <N> -- Step the program for N instructions
continue -- Continue program execution without stepping
print $N -- Print register N
print $fN -- Print floating point register N
print ADDR -- Print contents of memory at ADDRESS
print PC -- Print address of current instruction (not yet executed)
reinitialize -- Clear the memory and registers
breakpoint <ADDR> -- Set a breakpoint at address
delete <ADDR> -- Delete all breakpoints at address
list -- List all breakpoints
. -- Rest of line is assembly instruction to put in memory
<cr> -- Newline reexecutes previous command
? -- Print this message

Most commands can be abbreviated to their unique prefix
e.g., ex, re, l, ru, s, p

Programming Style Guidelines

Your projects will be graded on both correctness and clarity. Since you are not the only person who has to read your programs, you must comment them adequately so other people can understand them easily.

1) Program header

All your projects must start with a program header, something like this (fill in the information):

```
#
#       CSc 256 Project ??
#       Name: ????
#       Date: ????
#       Description: This program does ????
#
```

The Description should be a brief statement of what the program does, what the user has to enter (if anything), and what results the program prints out.

2) Indentation

Your MIPS assembly language program should be neatly divided into four columns which line up properly. (Hint: using "tabs" is a good way to line things up.) The first column should contain labels only. The second column should contain instruction operations only. The third column should contain the list of operands for that instruction. The fourth column should contain comments (if any).

Make sure that each line of your program code is less than 80 characters. Tabs should be set to 8 characters.

3) Commenting style

You should write your comments so that even someone who is not familiar with MIPS assembly language can follow what is happening in the program.

Describe all important variables used in the .data section. Identify which registers are used as which variables, where appropriate.

Describe what the instructions are doing within the context of the program, *not* the definition for each instruction. You don't have to comment every single instruction, but it should be clear what those lines of your program are doing. For example, this is good commenting style:

```
# i     $s0
# x     $s1
# sum   $s2

main:   li      $s0,1           # for (i=1; i<=10;i++){

loop:   li      $v0,5           #    cin >> x;
        syscall
        move    $s1,$v0
        blez    $s1,skip        #    if (x > 0)
        add     $s2,$s2,$s0     #        sum=sum+i;
skip:   add     $s0,$s0,1       # }
        ble     $s0,10,loop
```

Notice that C/C++ style indentations within the comments helps to make the comments more readable. Just by reading the comments, we can tell that the "sum=sum+i" line is executed if x > 0, and all the statements from "cin..." through "sum = ..." belong inside the for loop.

In contrast, this is much less readable:

```
loop:   li      $v0,5           # cin >> x;
        syscall
        move    $s1,$v0
        blez    $s1,skip        # if (x <= 0) goto skip
        add     $s2,$s2,$s0     # sum=sum+i;
skip:   add     $s0,$s0,1       # i++;
        ble     $s0,10,loop     # if (i <= 10) goto loop
```

Almost every single instruction is commented, but it's harder to read the comments. It's not obvious that we have a loop, and what goes inside the loop body.

The programming examples that you'll use in this class, in ~whsu/256/PROGS, will give you many examples of what the instructor considers good commenting style. (However, the programs provided in the lab exercises are often not properly commented; the difference should be obvious.)

ASCII Codes

ASCII code (hex)	char	ASCII code (hex)	char	ASCII code (hex)	char
0	nul				
7	bel				
a	nl				
d	cr				
20	sp	40	@	60	`
21	!	41	A	61	a
22	"	42	B	62	b
23	#	43	C	63	c
24	$	44	D	64	d
25	%	45	E	65	e
26	&	46	F	66	f
27	'	47	G	67	g
28	(	48	H	68	h
29	)	49	I	69	i
2a	*	4a	J	6a	j
2b	+	4b	K	6b	k
2c	,	4c	L	6c	l
2d	-	4d	M	6d	m
2e	.	4e	N	6e	n
2f	/	4f	O	6f	o
30	0	50	P	70	p
31	1	51	Q	71	q
32	2	52	R	72	r
33	3	53	S	73	s
34	4	54	T	74	t
35	5	55	U	75	u
36	6	56	V	76	v
37	7	57	W	77	w
38	8	58	X	78	x
39	9	59	Y	79	y
3a	:	5a	Z	7a	z
3b	;	5b	[	7b	{
3c	<	5c	\	7c	|
3d	=	5d	]	7d	}
3e	>	5e	^	7e	~
3f	?	5f	_	7f	del