
Instructions: Language of the Computer

The Stored Program Concept

The stored program concept says that the program is stored with data in the computer's memory. The computer is able to manipulate it as data—for example, to load it from disk, move it in memory, and store it back on disk.

- ❑ It is the basic operating principle for every computer.
- ❑ It is so common that it is taken for granted.
- ❑ Without it, every instruction would have to be initiated manually.

The Fetch-Execute Cycle

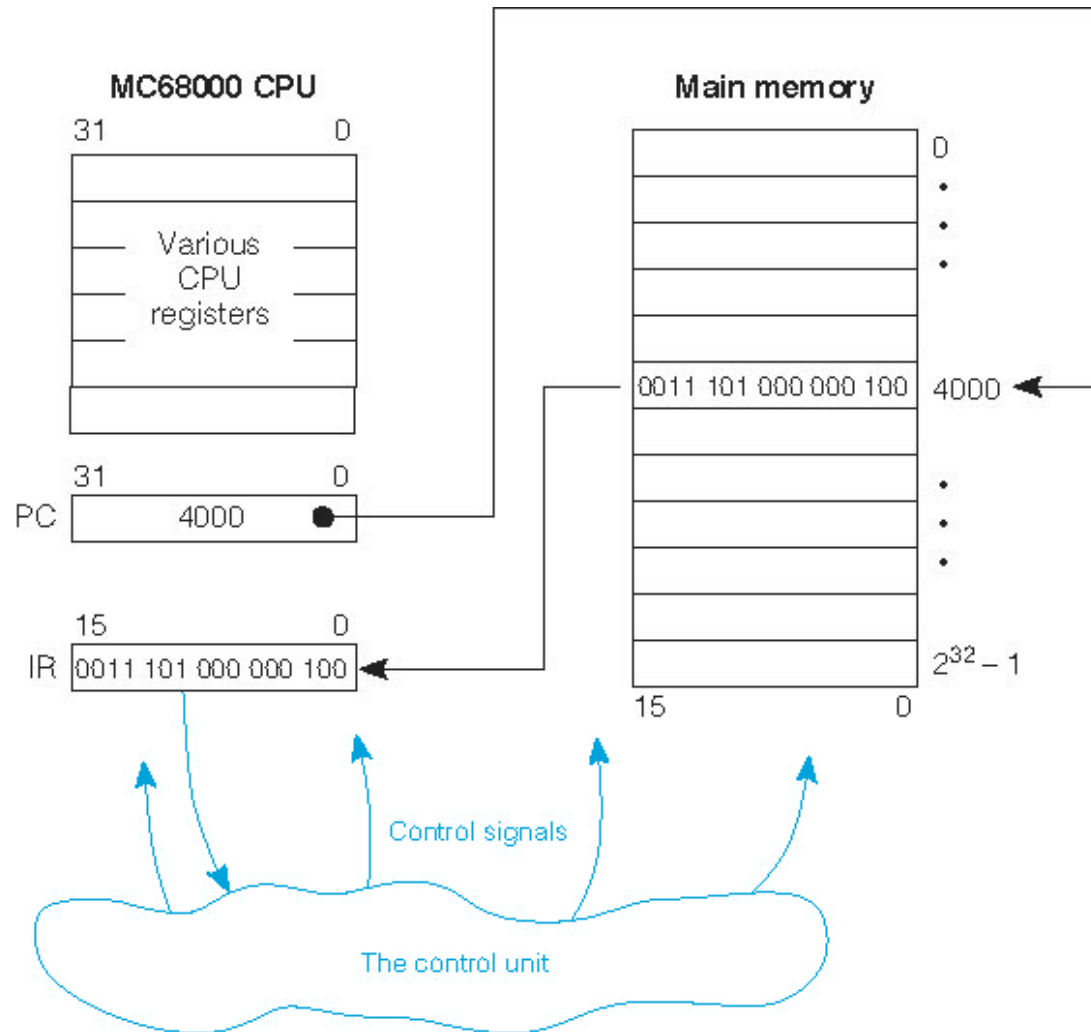
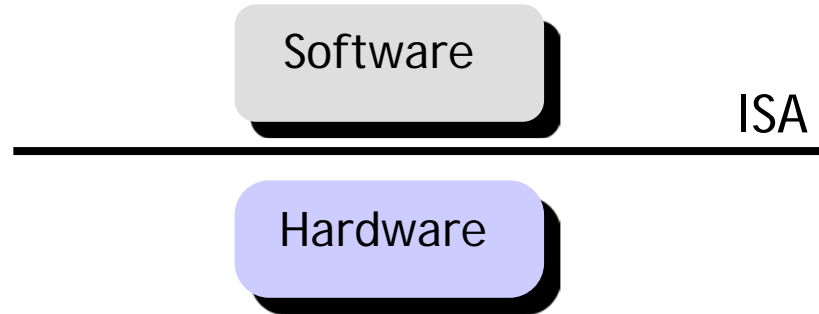


Fig. 1.2

Machine, Processor, and Memory State

- ❑ The **Machine State**: contents of all registers in system, accessible to programmer or not
- ❑ The **Processor State**: registers internal to the CPU
- ❑ The **Memory State**: contents of the memory system
- ❑ “State” is used in the formal finite state machine sense
- ❑ Maintaining or restoring the machine and processor state is important to many operations, especially procedure calls and interrupts

Instruction set architecture (ISA)



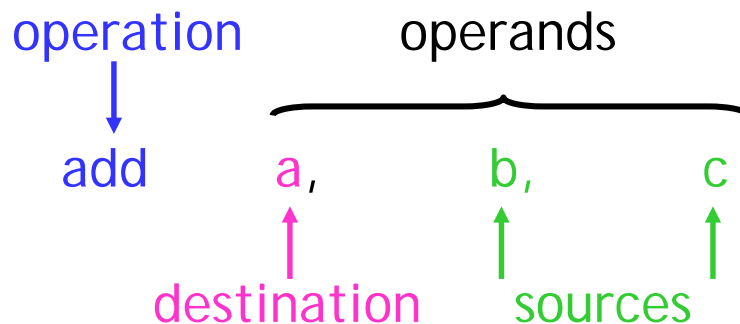
MIPS

- ❑ In this class, we'll use the MIPS instruction set architecture (ISA) to illustrate concepts in assembly language and machine organization
 - Of course, the concepts are not MIPS-specific
 - MIPS is just convenient because it is real, yet simple (unlike x86)
- ❑ The MIPS ISA is still used in many places today. Primarily in embedded systems, like:
 - Various routers from Cisco
 - Game machines like the Nintendo 64 and Sony Playstation 2
- ❑ You must become “fluent” in MIPS assembly:
 - Translate from C to MIPS and MIPS to C



MIPS: register-to-register, three address

- ❑ MIPS is a **register-to-register**, or **load/store**, architecture.
 - The destination and sources must all be registers.
 - Special instructions, which we'll see later, are needed to access main memory.
- ❑ MIPS uses **three-address** instructions for data manipulation.
 - Each ALU instruction contains a **destination** and two **sources**.
 - For example, an addition instruction ($a = b + c$) has the form:



MIPS register names

- ❑ MIPS register names begin with a \$. There are two naming conventions:

- By number:

\$0 \$1 \$2 ... \$31

- By (mostly) two-character names, such as:

\$a0-\$a3 \$s0-\$s7 \$t0-\$t9 \$sp \$ra

- ❑ Not all of the registers are equivalent:

- E.g., register \$0 or \$zero always contains the value 0
 - (go ahead, try to change it)

- ❑ Other registers have special uses, by convention:

- E.g., register \$sp is used to hold the “stack pointer”

- ❑ You have to be a little careful in picking registers for your programs.

Policy of Use Conventions

Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	Saved temporaries
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	reserved for OS kernel
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Basic arithmetic and logic operations

- ❑ The basic integer arithmetic operations include the following:

`add` `sub` `mul` `div`

- ❑ And here are a few logical operations:

`and` `or` `xor`

- ❑ Remember that these all require three register operands; for example:

```
add    $t0, $t1, $t2      # $t0 = $t1 + $t2
xor     $s1, $s1, $a0      # $s1 = $s1 xor $a0
```

Immediate operands

- ❑ The ALU instructions we've seen so far expect register operands. How do you get data into registers in the first place?

- Some MIPS instructions allow you to specify a signed constant, or “immediate” value, for the second source instead of a register. For example, here is the immediate add instruction, **addi**:

```
addi    $t0, $t1, 4      # $t0 = $t1 + 4
```

- Immediate operands can be used in conjunction with the **\$zero** register to write constants into registers:

```
addi    $t0, $0, 4       # $t0 = 4
```

- Data can also be loaded first into the memory along with the executable file. Then you can use load instructions to put them into registers

```
lw      $t0, 8($t1)      # $t0 = mem[8+$t1]
```

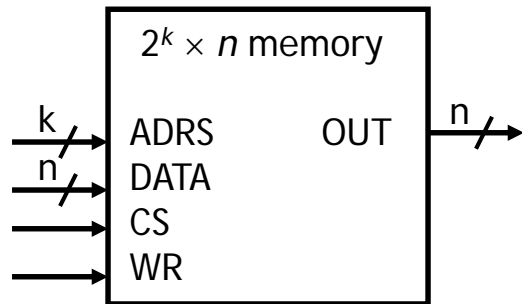
- ❑ MIPS is considered a **load/store** architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.

We need more space – memory

- ❑ **Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bit wide.**
 - That's not enough to hold data structures like large arrays.
 - We also can't access data elements that are wider than 32 bits.
- ❑ **We need to add some main memory to the system!**
 - RAM is cheaper and denser than registers, so we can add lots of it.
 - But memory is also significantly slower, so registers should be used whenever possible.
- ❑ **In the past, using registers wisely was the programmer's job.**
 - For example, C has a keyword “register” that marks commonly-used variables which should be kept in the register file if possible.
 - However, modern compilers do a pretty good job of using registers intelligently and minimizing RAM accesses.

Memory review

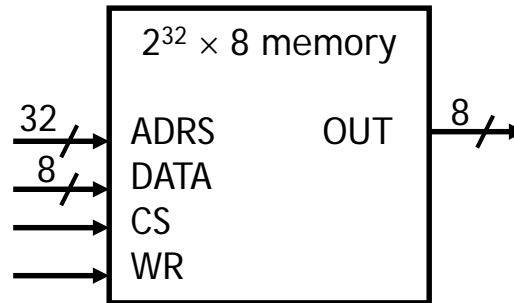
- ❑ Memory sizes are specified much like register files; here is a $2^k \times n$ bit RAM.



CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

- ❑ A chip select input **CS** enables or “disables” the RAM.
- ❑ **ADRS** specifies the memory location to access.
- ❑ **WR** selects between reading from or writing to the memory.
 - To read from memory, WR should be set to 0. **OUT** will be the n -bit value stored at ADRS.
 - To write to memory, we set $WR = 1$. **DATA** is the n -bit value to store in memory.

MIPS memory



- ❑ MIPS memory is **byte-addressable**, which means that each memory address references an 8-bit quantity.
- ❑ The MIPS architecture can support up to 32 address lines.
 - This results in a $2^{32} \times 8$ RAM, which would be 4 GB of memory.
 - Not all actual MIPS machines will have this much!

Bytes and words

- ❑ Remember to be careful with memory addresses when accessing words.
- ❑ For instance, assume an array of **words** begins at address 2000.
 - The first array element is at address 2000.
 - The second word is at address 2004, not 2001.
- ❑ For example, if \$a0 contains 2000, then

`lw $t0, 0($a0)`

accesses the first word of the array, but

`lw $t0, 8($a0)`

would access the **third** word of the array, at address 2008.

Loading and storing bytes

- ❑ The MIPS instruction set includes dedicated load and store instructions for accessing memory.
- ❑ The main difference is that MIPS uses **indexed addressing**.
 - The address operand specifies a signed constant and a register.
 - These values are added to generate the effective address.
- ❑ The MIPS “load byte” instruction **lb** transfers one byte of data from main memory to a register.


`lb $t0, 20($a0) # $t0 = Memory[$a0 + 20]`

question: what about the other 3 bytes in \$t0?

Sign extension!

- ❑ The “store byte” instruction **sb** transfers the lowest byte of data from a register into main memory.


`sb $t0, 20($a0) # Memory[$a0 + 20] = $t0`

Loading and storing words

- ❑ You can also load or store 32-bit quantities—a complete **word** instead of just a byte—with the **lw** and **sw** instructions.

```
lw $t0, 20($a0)      # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)      # Memory[$a0 + 20] = $t0
```

- ❑ Most programming languages support several 32-bit data types.
 - Integers
 - Single-precision floating-point numbers
 - Memory addresses, or pointers
- ❑ Unless otherwise stated, we'll assume words are the basic unit of data.

Computing with memory

- ❑ So, to compute with memory-based data, you must:
 1. Load the data from memory to the register file.
 2. Do the computation, leaving the result in a register.
 3. Store that value back to memory if needed.
- ❑ For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language using as few registers as possible?

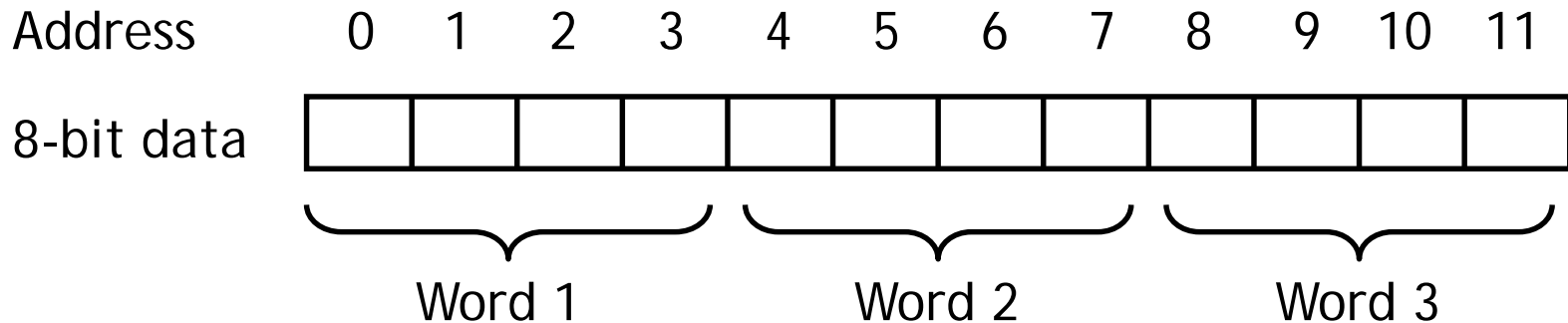
```
char A[4] = {1, 2, 3, 4};
```

```
int result;
```

```
result = A[0] + A[1] + A[2] + A[3];
```

Memory alignment


- ❑ Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations (bytes) of main memory.



- ❑ The MIPS architecture requires words to be **aligned** in memory; 32-bit words must start at an address that is divisible by 4.
 - 0, 4, 8 and 12 are valid **word addresses**.
 - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
 - Unaligned memory accesses result in a **bus error**, which you may have unfortunately seen before.
- ❑ This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

Exercise

□ Can we figure out the code?

<pre>swap(int v[], int k); { int temp; temp = v[k] v[k] = v[k+1]; v[k+1] = temp; }</pre>		<pre>swap: ; \$5=k \$4=v[0] sll \$2, \$5, 2; \$2←k×4 add \$2, \$4, \$2; \$2←v[k] lw \$15, 0(\$2) ; \$15←v[k] lw \$16, 4(\$2) ; \$16←v[k+1] sw \$16, 0(\$2) ; v[k]←\$16 sw \$15, 4(\$2) ; v[k+1]←\$15 jr \$31 Assuming k is stored in \$5, and the starting address of v[] is in \$4.</pre>
--	---	---

Pseudo-instructions

- ❑ MIPS assemblers support **pseudo-instructions** that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, “real” instructions.
- ❑ For example, you can use the **li** and **move** pseudo-instructions:

```
li      $a0, 2000      # Load immediate 2000 into $a0
move    $a1, $t0       # Copy $t0 into $a1
```

- ❑ They are probably clearer than their corresponding MIPS instructions:

```
addi    $a0, $0, 2000  # Initialize $a0 to 2000
add     $a1, $t0, $0    # Copy $t0 into $a1
```

- ❑ We'll see lots more pseudo-instructions this semester.
 - A core instruction set is given in “Green Card” of the text (J. Hennessy and D. Patterson, 1st page).
 - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

Control flow in high-level languages

- ❑ The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- ❑ **Conditional statements** execute only if some test expression is true.

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
    v0 = -v0;           // This might not be executed
v1 = v0 + v0;
```

- ❑ **Loops** cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];    // These statements will
    t0++;               // be executed five times
}
```

MIPS control instructions

- ❑ In this lecture, we introduced some of MIPS's control-flow instructions

<code>j immediate</code>	<code>// for unconditional jumps</code>
<code>jr \$r1</code>	<code>// jump to address stored in \$r1</code>
<code>bne and beq \$r1, \$r2, label</code>	<code>// for conditional branches</code>
<code>slt and slti \$rd, \$rs, \$rt</code>	<code>// set if less than (w/ and w/o an immediate)</code>
<code>\$rs, \$rt, imm</code>	

- ❑ And how to implement loops

- ❑ Today, we'll talk about
 - MIPS's pseudo branches
 - if/else
 - case/switch

Pseudo-branches

- ❑ The MIPS processor only supports two branch instructions, **beq** and **bne**, but to simplify your life the assembler provides the following other branches:

```
blt    $t0, $t1, L1    // Branch if $t0 < $t1
ble    $t0, $t1, L2    // Branch if $t0 <= $t1
bgt    $t0, $t1, L3    // Branch if $t0 > $t1
bge    $t0, $t1, L4    // Branch if $t0 >= $t1
```

- ❑ Later this term we'll see how supporting just **beq** and **bne** simplifies the processor design.

Implementing pseudo-branches

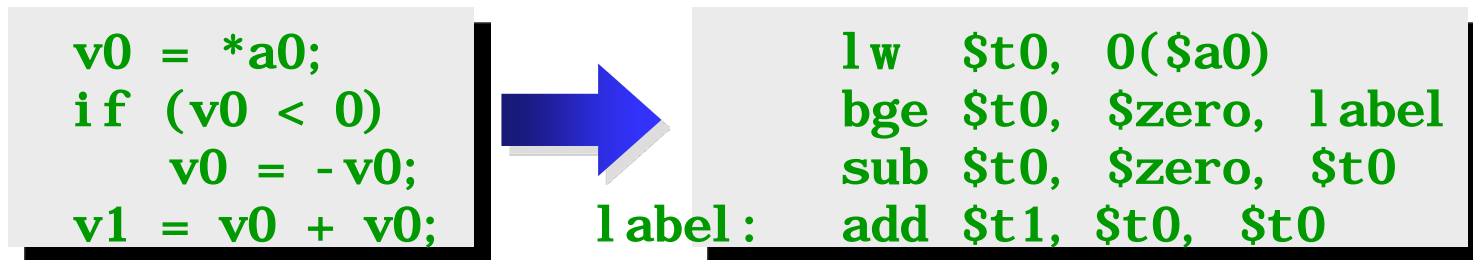
- ❑ Most pseudo-branches are implemented using `slt`. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt    $at, $a0, $a1        // $at = 1 if $a0 < $a1
bne    $at, $0, Label       // Branch if $at != 0
```

- ❑ Can you translate other pseudo-branches?
- ❑ All of the pseudo-branches need a register to save the result of `slt`, even though it's not needed afterwards.
 - MIPS assemblers use register `$1`, or `$at`, for temporary storage.
 - You should be careful in using `$at` in your own programs, as it may be **overwritten** by assembler-generated code.

Translating an if-then statement

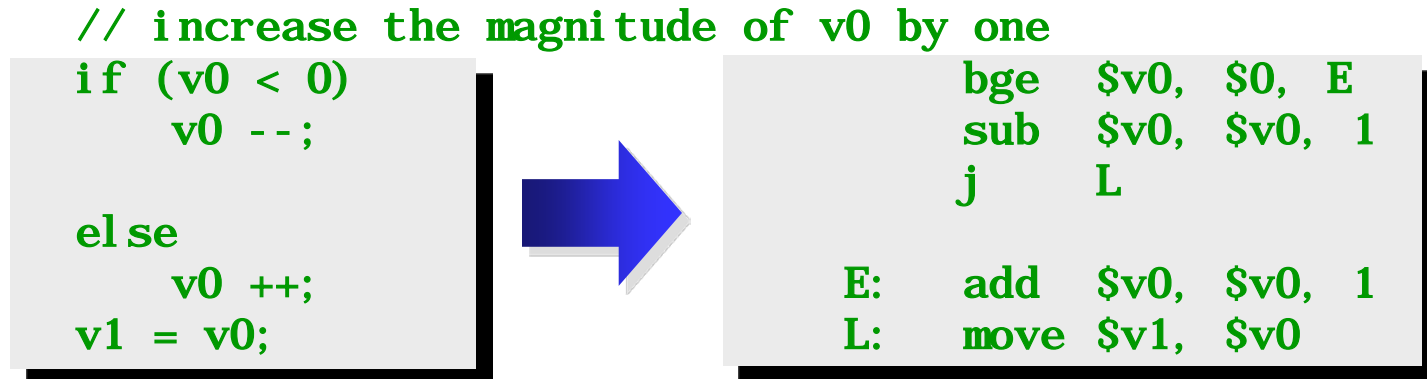
- We can use branch instructions to translate if-then statements into MIPS assembly code.



- Sometimes it's easier to *invert* the original condition.
 - In this case, we changed “continue if $v0 < 0$ ” to “skip if $v0 \geq 0$ ”.
 - This saves a few instructions in the resulting assembly code.

Translating an if-then-else statements

- ❑ If there is an **else** clause, it is the target of the conditional branch
 - And the **then** clause needs a jump over the **else** clause



- ❑ Dealing with else-if code is similar, but the target of the first branch will be another if statement.
 - Drawing the control-flow graph can help you out.

Example of a Loop Structure

```
for (i=1000; i>0; i--)  
    x[i] = x[i] + h;
```



```
Loop: lw  $s0, 0($s1)      ;$s1=x[1000]  
      add $s3, $s0, $s2    ;$s2=h  
      sw  $s3, 0($s1)  
      addi $s1, $s1, # - 4  
      bne $s1, $s5, Loop   ;$s5=x[0]
```

Assume: addresses of
x[1000] and x[0]
are in \$s1 and \$s5
respectively; h is
in \$s2;

1. How to initialize \$s5 ?
2. Can you use a counter to count the # of iterations instead of using \$s5? How many instructions are executed in this case?

Homework

- Let's write a program to count how many bits are zero in a 32-bit word. Suppose the word is stored in register \$t0.

- **C code**

```
int input, i, counter, bit, position;
counter = 0;
position = 1;
For (i=0; i<32; i++) {
    bit = input & position;
    if (bit == 0)
        counter++
    position = position << 1;
}
```

Functions calls in MIPS

- ❑ **We'll talk about the 3 steps in handling function calls:**
 1. The program's flow of control must be changed.
 2. Arguments and return values are passed back and forth.
 3. Local variables can be allocated and destroyed.

- ❑ **And how they are handled in MIPS:**
 - New instructions for calling functions.
 - Conventions for sharing registers between functions.
 - Use of a stack.

Control flow in C

- ❑ Invoking a function changes the control flow of a program twice.
 1. **Calling** the function
 2. **Returning** from the function
- ❑ In this example the **main** function calls **fact** twice, and fact returns twice—but to *different* locations in main.
- ❑ Each time fact is called, the CPU has to remember the appropriate **return address**.
- ❑ Notice that main itself is also a function! It is called by the operating system when you run the program.

```
int main()
{
    ...
    t1 = fact(8);
    t3 = t1 + t2;
    t2 = fact(3);
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    return f;
}
```

Control flow in MIPS

- ❑ MIPS uses the jump-and-link instruction **jal** to call functions.
 - The jal saves the return address (the address of the *next* instruction) in the dedicated register **\$ra**, before jumping to the function.
 - jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in \$ra.

jal Fact

- ❑ To transfer control back to the caller, the function just has to jump to the address that was stored in \$ra.

jr \$ra

- ❑ Let's now add the jal and jr instructions that are necessary for our factorial example.

Changing the control flow in MIPS

```
int main()
{
    ...
    jal Fact;
    ...
    t3 = t1 + t2;
    ...
    jal Fact;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    jr $ra;
}
```

Data flow in C

- ❑ Functions accept **arguments** and produce **return values**.
- ❑ The **black** parts of the program show the actual and formal arguments of the fact function.
- ❑ The **purple** parts of the code deal with returning and using a result.

```
int main()  
{  
    ...  
    t1 = fact(8);  
    t3 = t1 + t2;  
    t2 = fact(3);  
    ...  
}  
  
int fact(int n)  
{  
    int i, f = 1;  
    for (i = n; i > 1; i--)  
        f = f * i;  
    return f;  
}
```

Data flow in MIPS

- ❑ MIPS uses the following conventions for function arguments and results.
 - Up to four function arguments can be “passed” by placing them in argument registers **\$a0-\$a3** before calling the function with jal.
 - A function can “return” up to two values by placing them in registers **\$v0-\$v1**, before returning via jr.
- ❑ These conventions are not enforced by the hardware or assembler, but programmers agree to them so functions written by different people can interface with each other.
- ❑ Later we’ll talk about handling additional arguments or return values.

Nested functions

- ❑ What happens when you call a function that then calls another function?
- ❑ Let's say A calls B, which calls C.
 - The arguments for the call to C would be placed in \$a0-\$a3, thus *overwriting* the original arguments for B.
 - Similarly, **jal C** overwrites the return address that was saved in \$ra by the earlier **jal B**.

```
A:    ...  
      # Put B's args in $a0-$a3  
      jal B      # $ra = A2  
A2:   ...
```

```
B:    ...  
      # Put C's args in $a0-$a3,  
      # erasing B's args!  
      jal C      # $ra = B2  
B2:   ...  
      jr $ra     # Where does  
                # this go???
```

```
C:    ...  
      jr $ra
```

Spilling registers

- ❑ The CPU has a limited number of registers for use by all functions, and it's possible that several functions will need the same registers.
- ❑ We can keep important registers from being overwritten by a function call, by **saving** them before the function executes, and **restoring** them after the function completes.



- ❑ But there are two important questions.
 - Who is responsible for saving registers—the caller or the callee?
 - Where exactly are the register contents saved?

Who saves the registers?

- ❑ **However, in the typical “black box” programming approach, the caller and callee do not know anything about each other’s implementation.**
 - Different functions may be written by different people or companies.
 - A function should be able to interface with any client, and different implementations of the same function should be substitutable.
- ❑ **Who is responsible for saving important registers across function calls?**
 - The caller knows which registers are important to it and should be saved.
 - The callee knows exactly which registers it will use and potentially overwrite.
- ❑ **So how can two functions cooperate and share registers when they don’t know anything about each other?**

The caller could save the registers...

- ❑ One possibility is for the *caller* to save any important registers that it needs before making a function call, and to restore them after.
- ❑ But the caller does not know what registers are actually written by the function, so it may save more registers than necessary.
- ❑ In the example on the right, **frodo** wants to preserve **\$a0**, **\$a1**, **\$s0** and **\$s1** from **gollum**, but **gollum** may not even use those registers.

```
frodo: li    $a0, 3
        li    $a1, 1
        li    $s0, 4
        li    $s1, 1

        # Save registers
        # $a0, $a1, $s0, $s1

        jal   gollum

        # Restore registers
        # $a0, $a1, $s0, $s1

        add   $v0, $a0, $a1
        add   $v1, $s0, $s1
        jr    $ra
```

...or the callee could save the registers...

- ❑ Another possibility is if the *callee* saves and restores any registers it might overwrite.
- ❑ For instance, a **gollum** function that uses registers **\$a0**, **\$a2**, **\$s0** and **\$s2** could save the original values first, and restore them before returning.
- ❑ But the callee does not know what registers are important to the caller, so again it may save more registers than necessary.

gollum:

```
# Save registers  
# $a0 $a2 $s0 $s2
```

```
li    $a0, 2  
li    $a2, 7  
li    $s0, 1  
li    $s2, 8
```

```
...
```

```
# Restore registers  
# $a0 $a2 $s0 $s2
```

```
jr    $ra
```


...or they could work together

- ❑ MIPS uses conventions again to split the register spilling chores.
- ❑ The *caller* is responsible for saving and restoring any of the following **caller-saved registers** that it cares about.

\$t0-\$t9

\$a0-\$a3

\$v0-\$v1

In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.

- ❑ The *callee* is responsible for saving and restoring any of the following **callee-saved registers** that it uses. (Remember that \$ra is “used” by jal.)

\$s0-\$s7

\$ra

Thus the caller may assume these registers are not changed by the callee.

- **\$ra** is tricky; it is saved by a callee who is also a caller.
- ❑ Be especially careful when writing nested functions, which act as both a caller and a callee!

Register spilling example

- This convention ensures that the caller and callee together save all of the important registers—frodo only needs to save registers **\$a0** and **\$a1**, while gollum only has to save registers **\$s0** and **\$s2**.

```
frodo:  li    $a0, 3
        li    $a1, 1
        li    $s0, 4
        li    $s1, 1

        # Save registers
        # $a0, $a1

        jal    gollum

        # Restore registers
        # $a0 and $a1

        add    $v0, $a0, $a1
        add    $v1, $s0, $s1

        jr     $ra
```

```
gollum:                                     # Save registers
                                           # $s0, $s2

        li    $a0, 2
        li    $a2, 7
        li    $s0, 1
        li    $s2, 8

        ...

        # Save $ra, $a0 & $a2
        jal    gollumson

        # Restore registers
        # $a0 & $a2

        ...

        # Restore $s0, $s2, $ra
        jr     $ra
```

Where are the registers saved?

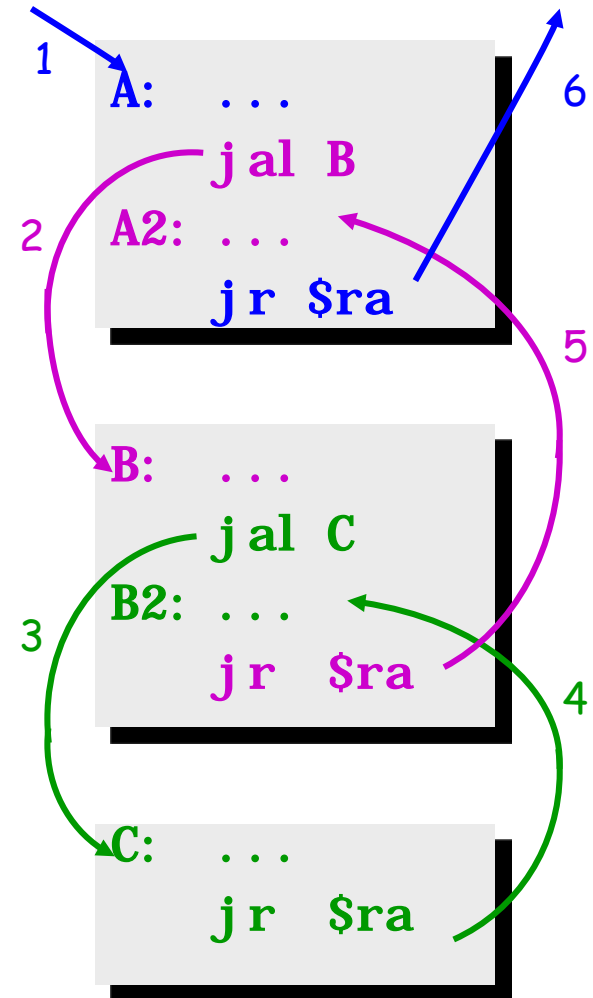
- ❑ Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.
- ❑ It would be nice if each function call had its own private memory area.
 - This would prevent other function calls from overwriting our saved registers.
 - We could use this private memory for other purposes too, like storing local variables.

Function calls and stacks

- ❑ Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns

- ❑ Here, for example, C must return to B *before* B can return to A.



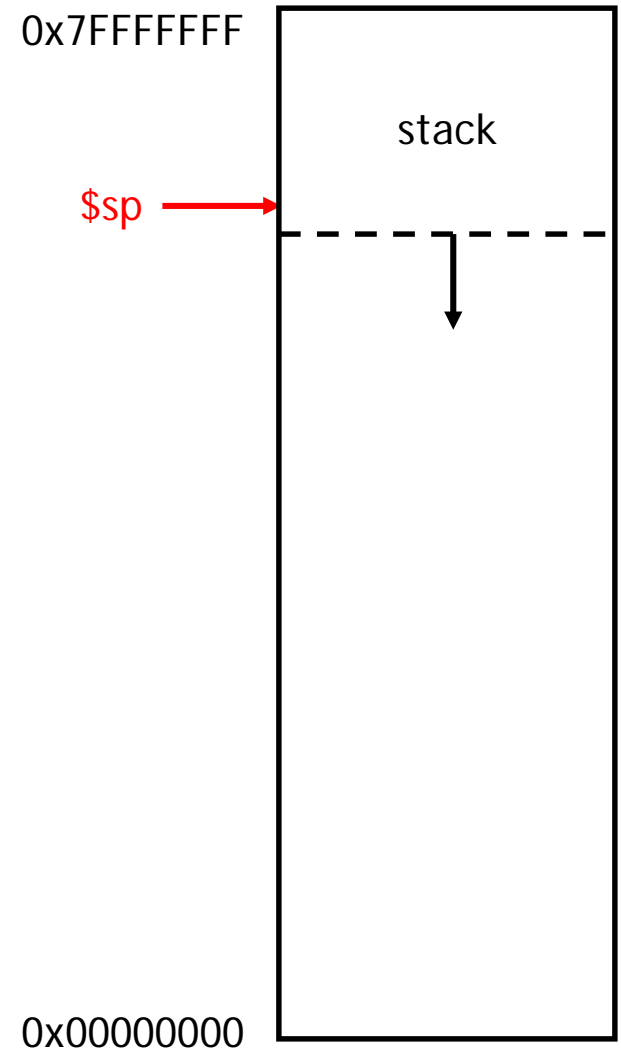
Program Stack

- ❑ It's natural to use a **stack** for function call storage. A block of stack space, called a **stack frame**, can be allocated for each function call.
 - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
 - Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- ❑ The stack frame (so called “activation frame” or “activation record”) can be used for several purposes.
 - Caller- and callee-**save registers** can be put in the stack.
 - The stack frame can also hold **local variables** such as arrays, or **extra arguments** and **return values**.
- ❑ Prologue (procedure entrance): Allocates an activation frame on the stack
- ❑ Epilogue (exit from procedure): De-allocates the frame, does actual return



The MIPS stack

- ❑ In MIPS machines, part of main memory is reserved for a stack.
 - The stack grows downward in terms of memory addresses.
 - The address of the top element of the stack is stored (by convention) in the “stack pointer” register, **\$sp (\$29)**.
- ❑ MIPS does not provide “push” and “pop” instructions. Instead, they must be done explicitly by the programmer.



Pushing elements

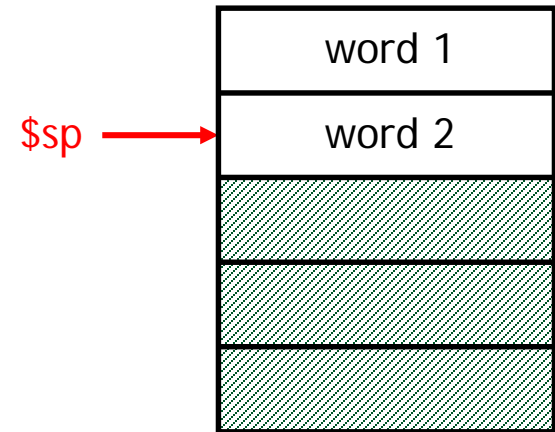
- ❑ To **push** elements onto the stack:
 - Move the stack pointer **\$sp** down to make room for the new data.
 - Store the elements into the stack.
- ❑ For example, to push registers **\$t1** and **\$t2** onto the stack:

```
sub  $sp, $sp, 8
sw   $t1, 4($sp)
sw   $t2, 0($sp)
```

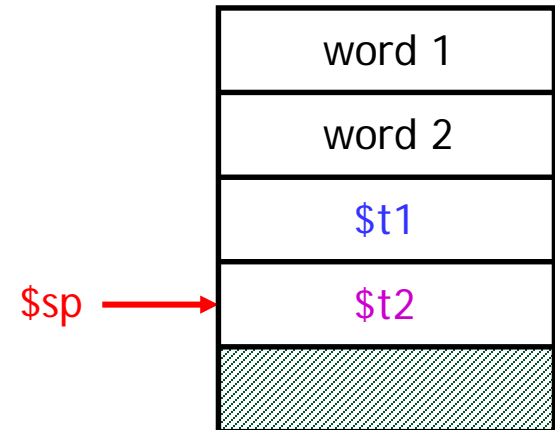
- ❑ An equivalent sequence is:

```
sw   $t1, -4($sp)
sw   $t2, -8($sp)
sub  $sp, $sp, 8
```

- ❑ Before and after diagrams of the stack are shown on the right.



Before



After

Accessing and popping elements

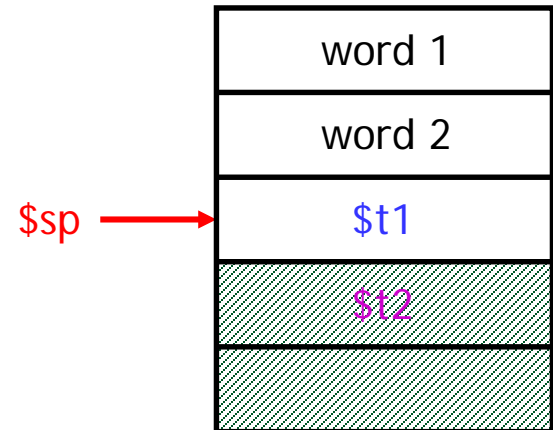
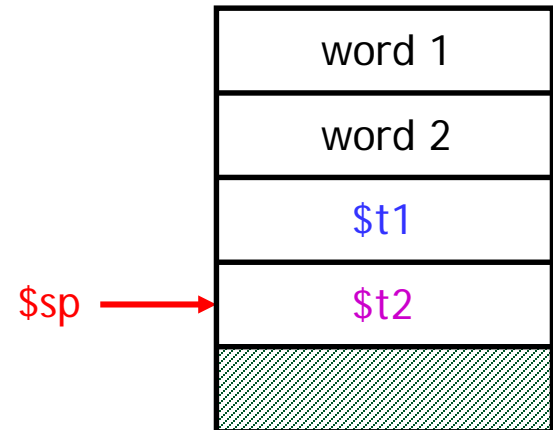
- ❑ You can access any element in the stack (not just the top one) if you know where it is relative to `$sp`.
- ❑ For example, to retrieve the value of `$t1`:

```
lw    $s0, 4($sp)
```

- ❑ You can **pop**, or “erase,” elements simply by adjusting the stack pointer upwards.
- ❑ To pop the value of `$t2`, yielding the stack shown at the bottom:

```
addi  $sp, $sp, 4
```

- ❑ Note that the popped data is still present in memory, but data past the stack pointer is considered invalid.



Function Call Example: Combinations

- ❑ A mathematical definition of combinations is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- ❑ The corresponding C code is below

```
int comb (int n, int k)
{
    return fact (n) / fact (k) / fact (n-k);
}
```

MIPS Code for Combinations

1 comb:			# n is in \$a0, k is in \$a1, will put result in \$v0
2	sub	\$sp, \$sp, 16	# prepare to push 4 words onto stack
3	sw	\$ra, 0(\$sp)	# comb is a callee, so save \$ra
4	sw	\$s0, 4(\$sp)	# save \$s0 coz it'll be used in line 8, 11, 16
5	sw	\$a0, 8(\$sp)	# comb is a caller in relation to fact so save \$a0
6	sw	\$a1, 12(\$sp)	# ...and \$a1 as they are used in 9, 12, 13, 14
7	jal	fact	# compute n!, its arg is in \$a0 already
8	move	\$s0, \$v0	# fact returns result (n!) in \$v0, save it in \$s0
9	lw	\$a0, 12(\$sp)	# set \$a0 with k on the stack
10	jal	fact	# compute k!
11	div	\$s0, \$s0, \$v0	# fact returns result in \$v0, compute $n!/k! \Rightarrow \$s0$
12	lw	\$a0, 8(\$sp)	# load n from stack
13	lw	\$a1, 12(\$sp)	# load k from stack
14	sub	\$a0, \$a0, \$a1	# set \$a0 with n-k
15	jal	fact	# compute (n-k)!
16	div	\$s0, \$s0, \$v0	# compute $n!/k!/(n-k)!$. This is the result
17	move	\$v0, \$s0	# prepare to return. 1. Put result in \$v0
18	lw	\$ra, 0(\$sp)	# 2. restore \$ra
19	lw	\$s0, 4(\$sp)	# ... and \$s0
20	addi	\$sp, \$sp, 16	# 3. deallocate stack frame
21	jr	\$ra	# does the actual return

Summary

- ❑ Today we focused on implementing function calls in MIPS.
 - We call functions using **jal**, passing arguments in registers **\$a0-\$a3**.
 - Functions place results in **\$v0-\$v1** and return using **jr \$ra**.
- ❑ Managing resources is an important part of function calls.
 - To keep important data from being overwritten, registers are saved according to conventions for **caller-save** and **callee-save** registers.
 - Each function call uses stack memory for saving registers, storing local variables and passing extra arguments and return values.
- ❑ Assembly programmers must follow many conventions. Nothing prevents a rogue program from overwriting registers or stack memory used by some other function.

Assembly vs. machine language

- ❑ So far we've been using **assembly language**.
 - We assign names to operations (e.g., **add**) and operands (e.g., **\$t0**).
 - Branches and jumps use labels instead of actual addresses.
 - Assemblers support many pseudo-instructions.
- ❑ Programs must eventually be translated into **machine language**, a binary format that can be stored in memory and decoded by the CPU.
- ❑ MIPS machine language is designed to be easy to decode.
 - Each MIPS instruction is the same length, 32 bits.
 - There are only three different instruction formats, which are very similar to each other.
- ❑ Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

Three MIPS formats

- ❑ simple instructions all 32 bits wide
- ❑ very structured, no unnecessary baggage
- ❑ only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Signed value
-32768 ~ +32767

R-type: ALU instructions (add, sub,...)

*I-type: immediate (addi ...), loads (lw ...), stores (sw ...),
conditional branches (bne ...), jump register (jr ...)

J-type: jump (j), jump and link (jal)

Constants

- ❑ Small constants are used quite frequently (50% of operands)

e.g., $A = A + 5;$
 $B = B + 1;$
 $C = C - 18;$

- ❑ MIPS Instructions:

```
addi $29, $29, 4  
slti $8, $18, 10  
andi $29, $29, 6  
ori  $29, $29, 4
```

Larger constants

- ❑ Larger constants can be loaded into a register 16 bits at a time.
 - The load upper immediate instruction **lui** loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
 - An immediate logical OR, **ori**, then sets the lower 16 bits.
- ❑ To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui $s0, 0x003D          # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900      # $s0 = 003D 0900
```

- ❑ This illustrates the principle of making the common case fast.
 - Most of the time, 16-bit constants are enough.
 - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.
- ❑ Pseudo-instructions may contain large constants. Assemblers will translate such instructions correctly.
- ❑ We used a **lw** instruction before.

Loads and stores

- ❑ The limited 16-bit constant can present difficulties for accesses to global data.
 - Let's assume the assembler puts a variable at address 0x10010004.
 - 0x10010004 is bigger than 32,767
- ❑ In these situations, the assembler breaks the immediate into two pieces.

```
lui    $t0, 0x1001          # 0x1001 0000
lw     $t1, 0x0004($t0)     # Read from Mem[0x1001 0004]
```


Branches

- ❑ For branch instructions, the constant field is not an address, but an *offset* from the next program counter (PC+4) to the target address.

```
        beq    $a0, $0, L
        add    $v1, $v0, $0
        add    $v1, $v1, $v1
        j      Somewhere
L:      add    $v1, $v0, $v0
```

- ❑ Since the branch target L is three *instructions* past the first **add**, the address field would contain $3 \times 4 = 12$. The whole **beq** instruction would be stored as:

000100	00001	00000	0000 0000 0000 1100
op	rs	rt	address

Why (PC+4)? Will be clear when we learned pipelining

Larger branch constants

- ❑ Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- ❑ If you do need to branch further, you can use a jump with a branch. For example, if “Far” is very far away, then the effect of:

```
beq  $s0, $s1, Far  
...
```

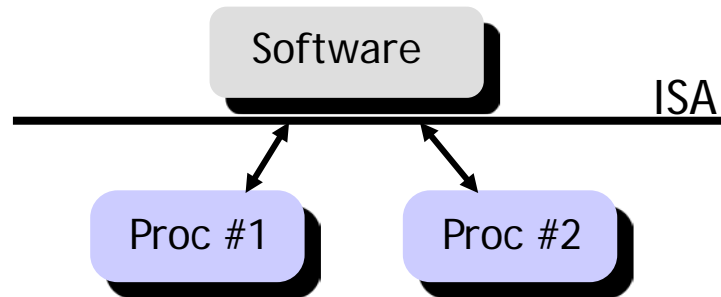
can be simulated with the following actual code.

```
                bne  $s0, $s1, Next  
                j    Far  
Next:           ...
```

- ❑ Again, the MIPS designers have taken care of the common case first.

Summary Instruction Set Architecture (ISA)

- ❑ The ISA is the interface between hardware and software.
- ❑ The ISA serves as an **abstraction layer** between the HW and SW
 - Software doesn't need to know how the processor is implemented
 - Any processor that implements the ISA appears equivalent



- ❑ An ISA enables processor innovation without changing software
 - This is how Intel has made billions of dollars.
- ❑ Before ISAs, software was re-written for each new machine.

RISC vs. CISC

- ❑ MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy, one of the authors of our textbook.
- ❑ The architecture is similar to that of other RISC architectures, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors.
- ❑ Older processors used **complex instruction sets computing, or CISC architectures**.
 - Many powerful instructions were supported, making the assembly language programmer's job much easier.
 - But this meant that the processor was more complex, which made the hardware designer's life harder.
- ❑ Many new processors use **reduced instruction sets computing, or RISC architectures**.
 - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
 - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- ❑ Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.

RISC vs. CISC

❑ Characteristics of ISAs

CISC	RISC
Variable length instruction	Single word instruction
Variable format	Fixed-field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

A little ISA history

- ❑ **1964: IBM System/360, the first computer family**
 - IBM wanted to sell a range of machines that ran the same software
- ❑ **1960's, 1970's: Complex Instruction Set Computer (CISC) era**
 - Much assembly programming, compiler technology immature
 - Simple machine implementations
 - Complex instructions simplified programming, little impact on design
- ❑ **1980's: Reduced Instruction Set Computer (RISC) era**
 - Most programming in high-level languages, mature compilers
 - Aggressive machine implementations
 - Simpler, cleaner ISA's facilitated pipelining, high clock frequencies
- ❑ **1990's: Post-RISC era**
 - ISA complexity largely relegated to non-issue
 - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
 - ISA compatibility outweighs any RISC advantage in general purpose
 - Embedded processors prefer RISC for lower power, cost
- ❑ **2000's: ??? EPIC? Dynamic Translation?**

Assessing and Understanding Performance

Why know about performance

❑ Purchasing Perspective:

- **Given a collection of machines, which has the**
 - Best Performance?
 - Lowest Price?
 - Best Performance/Price?

❑ Design Perspective:

- **Faced with design options, which has the**
 - Best Performance Improvement?
 - Lowest Cost?
 - Best Performance/Cost ?

❑ Both require

- **Metric for evaluation**
- **Basis for comparison**

Execution Time

❑ Elapsed Time

- counts everything (disk, I/O , etc.)
- a useful number, but often not good for comparison purposes
- can be broken up into system time, and user time

❑ CPU time

- doesn't count I/O or time spent running other programs
- Include memory accesses

❑ Our focus: user CPU time

- time spent executing the lines of code that are "in" our program

Book's Definition of Performance

- ❑ For some program running on machine X,

$$\text{Performance}_x = 1 / \text{Execution time}_x$$

- ❑ "X is n times faster than Y"

$$\text{Performance}_x / \text{Performance}_y = n$$

- ❑ **Problem:**

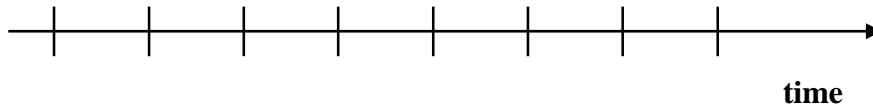
- machine A runs a program in 20 seconds
- machine B runs the same program in 25 seconds

Clock Cycles

- ❑ Instead of reporting execution time in seconds, we often use cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- ❑ Clock “ticks” indicate when to start activities (one abstraction):



- ❑ cycle time = time between ticks = seconds per cycle
- ❑ clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 200 Mhz. clock has a $\frac{1}{200 \times 10^6} \times 10^9 = 5$ nanoseconds cycle time

How to Improve Performance

□
$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- So, to improve performance (everything else being equal) you can either

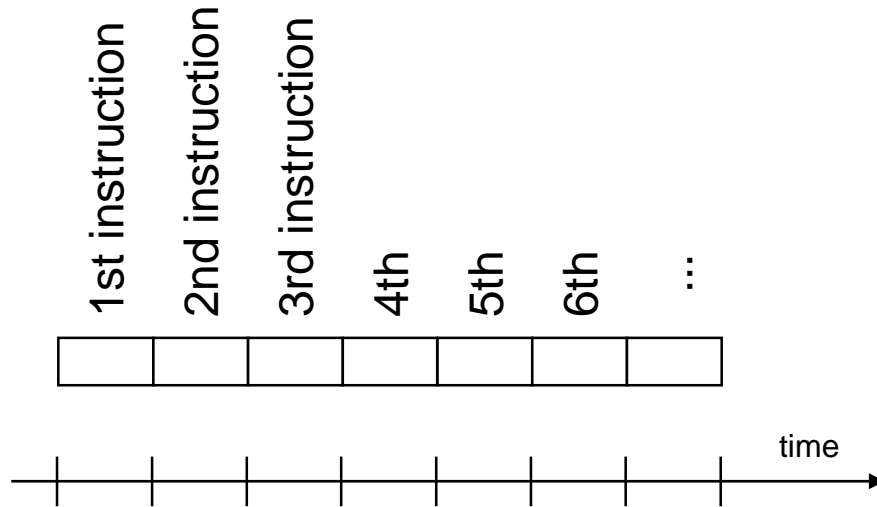
_____ ↓ the # of required cycles for a program, or

_____ ↓ the clock cycle time or, said another way,

_____ ↑ the clock rate.

How many cycles are for a program?

- ❑ Could assume that # of cycles = # of instructions

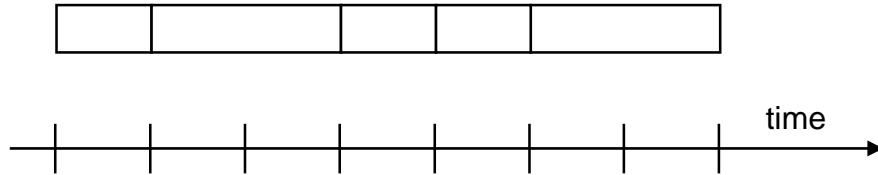


This assumption is incorrect,

different instructions take different amounts of time on different machines.

Why? hint: remember that these are machine instructions, not lines of C code

Different numbers of cycles for different instructions



- ☐ Multiplication takes more time than addition
- ☐ Floating point operations take longer than integer ones
- ☐ Accessing memory takes more time than accessing registers
- ☐ Important point: changing the cycle time often changes the number of cycles required for various instructions (more later)

Example

- Our favorite program runs in 10 seconds on computer A, which has a 400 Mhz. clock. We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program. What clock rate should we tell the designer to target?

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

For program A: 10 seconds = $\text{Cycles}_A \times 1/400\text{MHz}$

For program B: 6 seconds = $\text{Cycles}_B \times 1/\text{clock rate}_B$

$\text{Cycles}_B = 1.2 \text{ Cycles}_A$

$\text{Clock rate}_B = 800\text{MHz}$

Now that we understand cycles

- ❑ **A given program will require**
 - some number of instructions (machine instructions)
 - some number of cycles
 - some number of seconds
- ❑ **We have a vocabulary that relates these quantities:**
 - cycle time (seconds per cycle)
 - clock rate (cycles per second)
 - **CPI** (cycles per instruction)
 - a floating point intensive application might have a higher CPI
 - **MIPS** (millions of instructions per second)
 - this would be higher for a program using simple instructions

Another Way to Compute CPU Time

$$\text{CPU Time (or, Execution Time)} = \frac{\text{\# of instructions}}{\text{program}} \times \frac{\text{\# of cycles}}{\text{instruction}} \times \frac{\text{\# of seconds}}{\text{cycle}}$$

$$= \text{instruction count} \times \text{CPI} \times \text{cycle time}$$

$$= \text{instruction count} \times \text{CPI} \times \frac{1}{\text{clock rate}}$$

Performance

- ❑ Performance is determined by execution time
- ❑ Do any of the following variables alone equal performance?
 - # of cycles to execute program?
 - # of instructions in program?
 - # of cycles per second?
 - average # of cycles per instruction (CPI)?
 - average # of instructions per second?
- ❑ Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

CPI Example

- ❑ If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?
- ❑ Suppose we have two implementations of the same instruction set architecture (ISA).

For some program P,

Machine A has a clock cycle time of 10 ns. and a CPI of 2.0

Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

$$\text{CPU time}_A = \text{IC} \times \text{CPI} \times \text{cycle time} = \text{IC} \times 2.0 \times 10\text{ns} = 20 \times \text{IC ns}$$

$$\text{CPU time}_B = \text{IC} \times 1.2 \times 20\text{ns} = 24 \times \text{IC ns}$$

So, A is 1.2 (=24/20) times faster than B

What machine is faster for this program, and by how much?

of Instructions Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

Which sequence will be faster? How much? (assume CPU starts execute the 2nd instruction after the 1st one completes)
What is the CPI for each sequence?

$$\# \text{ of cycles}_1 = 2 \times 1 + 1 \times 2 + 2 \times 3 = 10$$

$$\# \text{ of cycles}_2 = 4 \times 1 + 1 \times 2 + 1 \times 3 = 9 \quad \text{So, sequence 2 is 1.1 times faster}$$

$$\text{CPI}_1 = 10 / 5 = 2$$

$$\text{CPI}_2 = 9 / 6 = 1.5$$

MIPS Example

- ❑ Two different compilers are being tested for a 100 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.

The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

- ❑ Which sequence will be faster according to MIPS?
- ❑ Which sequence will be faster according to execution time?

of instruction₁ = 5M + 1M + 1M = 7M, # of instruction₂ = 10M + 1M + 1M = 12M

of cycles₁ = 5M × 1 + 1M × 2 + 1M × 3 = 10M cycles = 0.1 seconds

of cycles₂ = 10M × 1 + 1M × 2 + 1M × 3 = 15M cycles = 0.15 seconds

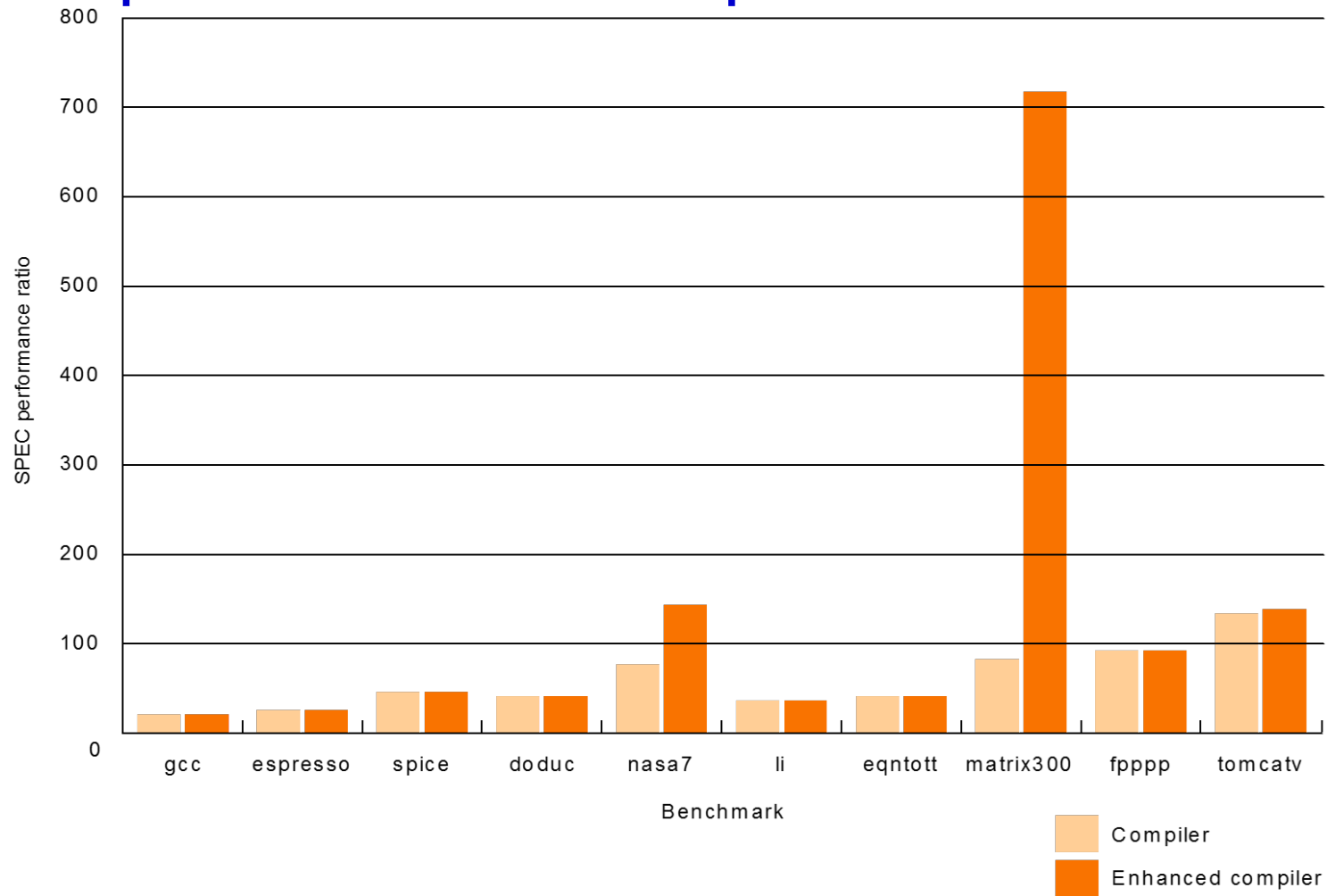
So, MIPS₁ = 7M/0.1 = 70MIPS, MIPS₂ = 12M/0.15 = 80MIPS > MIPS₁

Benchmarks

- ❑ **Performance best determined by running a real application**
 - Use programs typical of expected workload
 - Or, typical of expected class of applications
e.g., compilers/editors, scientific applications, graphics, etc.
- ❑ **Small benchmarks**
 - nice for architects and designers
 - easy to standardize
 - can be abused
- ❑ **SPEC (System Performance Evaluation Cooperative)**
 - companies have agreed on a set of real program and inputs
 - valuable indicator of performance (and compiler technology)
 - can still be abused

SPEC '89

❑ Compiler “enhancements” and performance



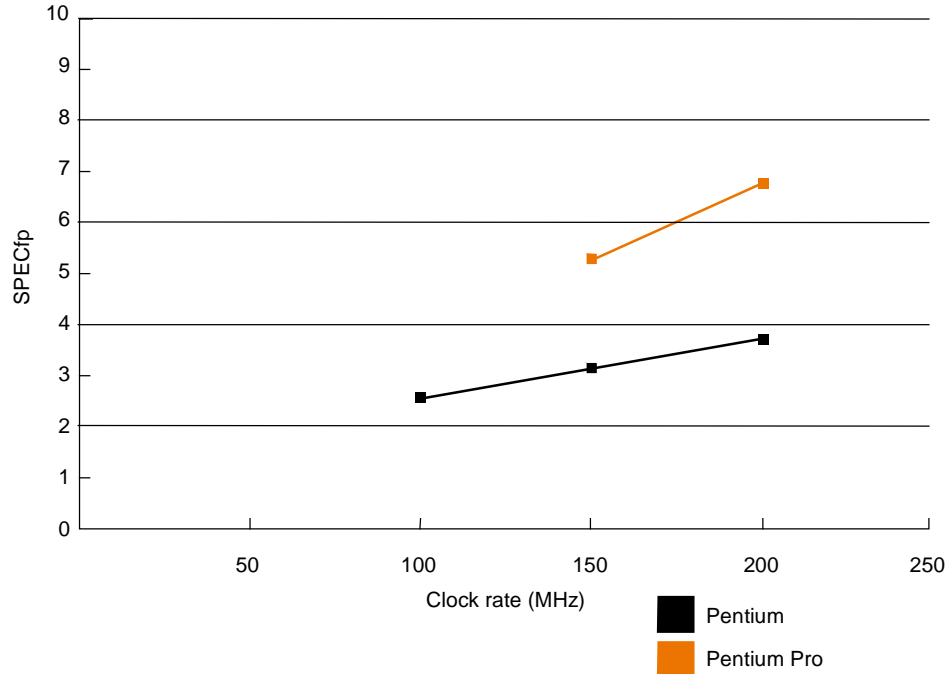
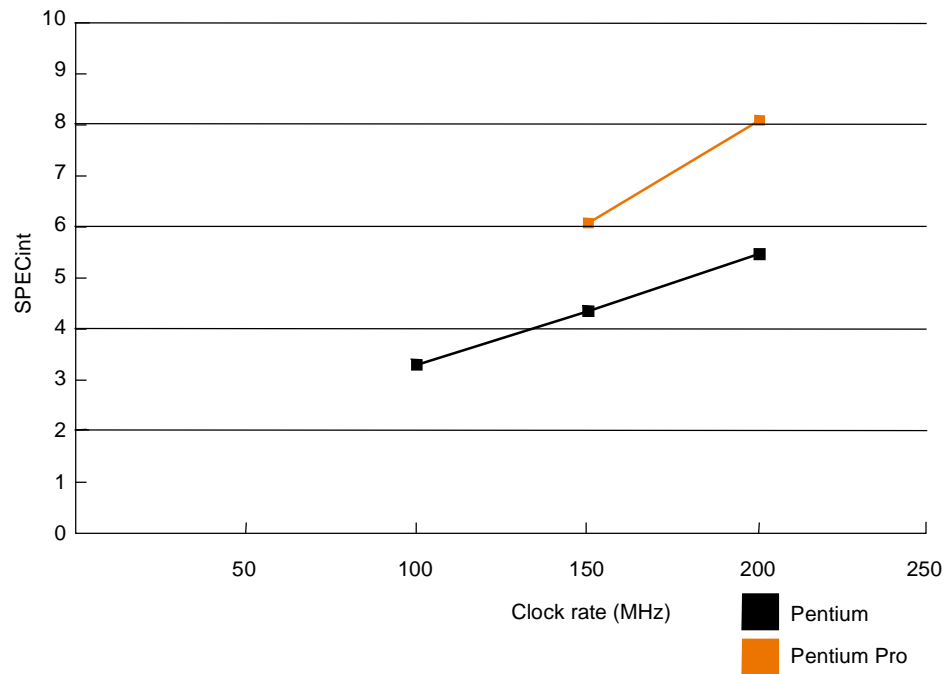
SPEC '95

Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
jpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Navier Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

SPEC '95

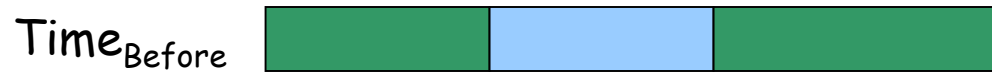
Does doubling the clock rate double the performance?

Can a machine with a slower clock rate have better performance?



Amdahl's Law

Execution Time After Improvement = Execution Time Unaffected + (Execution Time Affected / Amount of Improvement)



Execution time w/o E (Before)

Speedup (E) =

Execution time w E (After)

❑ **Example:**

"Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

How about making it 5 times faster?

❑ **Principle: Make the common case fast**

Example

- ❑ Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

$$10/6$$

- ❑ We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

$$100 - x + x/5 = 100/3, \quad x = 83.3$$

Remember

- ❑ **Performance is specific to a particular program/s**
 - Total execution time is a consistent summary of performance
- ❑ **For a given architecture performance increases come from:**
 - increases in clock rate (without adverse CPI affects)
 - improvements in processor organization that lower CPI
 - compiler enhancements that lower CPI and/or instruction count
- ❑ **Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance**