CSCI-2500: Computer Organization

Chapter 2: Instructions: "Lanaguage of the Computer"

Stored Program Computer

- Recall that computers read instructions from memory (memory is a big array of bits).
- Each instruction is represented by a *bunch* of bits.
- We can think of the program as input to the processor – each instruction tells the processor to perform some operations.

Processor Instruction Sets

- In general, a computer needs a few different kinds of instructions:
  - mathematical and logical operations
  - data movement (access memory)
  - jumping to new places in memory
    - if the right conditions hold.
  - I/O (sometimes treated as data movement)

Instruction Set Design

- An Instruction Set provides a *functional* description of a processor.
- It is the ***visible programmer interface*** to the processor.
- Question: how should we go about designing a general purpose instruction set?
  - by general purpose, we want any program to run on this instruction set and run as efficiently as possible.

Principle #1: Simplicity Favors Regularity

- We are already familiar with how to write standard symbolic equations such as:

  A = B + C;

  D = A – E;

- We would like our instruction set to look similar, particularly with respect to the number of operands.

Simplicity Favors Regularity (cont).

- Let's translate:

```
a = b + c;
d = a + e;
```

- In MIPS assembly language this is:

```
add a, b, c     # a = b + c
sub d, a, e     # d = a - e
```

- MIPS instruction only has two source operands and places the results in a destination operand.
- There are however a few exceptions to this rule....

Another example:

- Translate C language statement:
  f = (g + h) - (i + j);
- In MIPS:

```
add t0, g, h    # temp var t0 = g + h
add t1, i, j    # temp var t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

- # is a comment
- t0 and t1 are ***registers*** and serve as operands used by the processor.

Principal #2: Smaller is Faster

- Unlike high-level programming languages, instructions do not have access to a large number of variables.
- A small number of storage containers are provided within the processor to be used by instructions to read and write data.
- These storage containers are called ***registers***.
- There number is few because space on a processor is limited.
- Also, access time correlates to size, like a searching problem.
  - think about having to search thru 10 items vs. 1 million.

MIPS registers

- The MIPS processor has 32 general purpose registers (each is 32 bits wide).
- In MIPS assembly language the register names look like this:

  `$s0`, `$s1`, … and `$t0`, `$t1`, …

  We will find out why they are named like this a bit later.

MIPS Registers and 'C'

- For examples derived from 'C' code we will use:

  `$s0`, `$s1`, `$s2`, … for 'C' variables.

  `$t0`, `$t1`, `$t2`, … for temporary values.

Example: C to MIPS

$$\underset{\texttt{\$s0}}{\texttt{a}}\texttt{=(}\underset{\texttt{\$s1}}{\texttt{b}}\texttt{+}\underset{\texttt{\$s2}}{\texttt{c}}\texttt{)-(}\underset{\texttt{\$s3}}{\texttt{d}}\texttt{+}\underset{\texttt{\$s4}}{\texttt{e}}\texttt{);}$$

```
add $t0, $s1, $s2  # t0 = b+c
add $t1, $s3, $s4  # t1 = d+e
sub $s0, $t0, $t1  # a = $t0-$t1
```

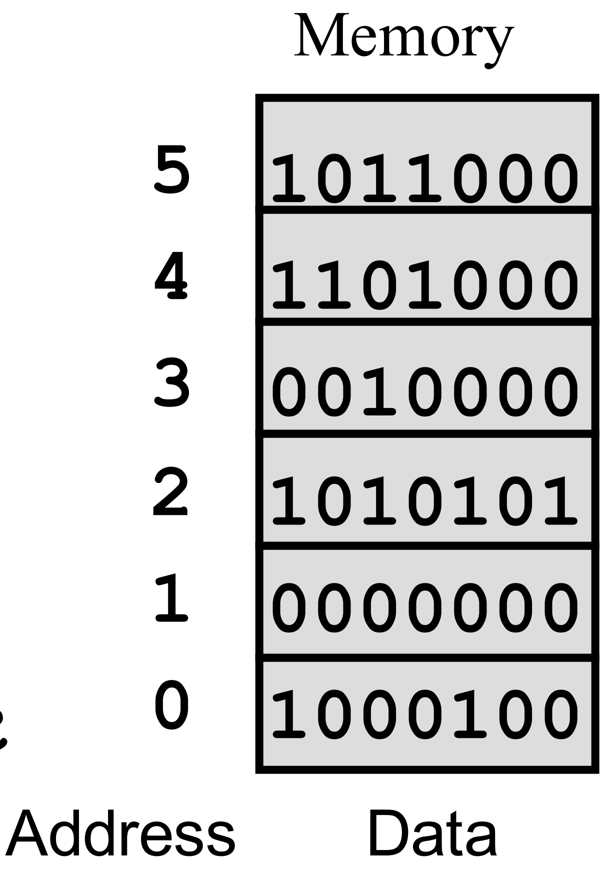
Registers vs. Memory

- In the MIPS instruction set, arithmetic operations occur *only* on registers.
- There may be more variables than registers.
- What about arrays?
- What about subroutines?
  - inside a subroutine we use different variables.

Data Transfer Instructions

- MIPS includes instructions that transfer data between registers and memory.

- To access some data in memory, we need to know the *address* of the data.



Bytes vs. Words

- MIPS registers are each 32 bits wide (1 word).
- Memory is organized in to 8-bit bytes.
- In the MIPS architecture, *words* must start at addresses that are a multiple of 4.
  - alignment restriction.
  - on 64 bit architectures, words are 8-byte aligned

Memory as Words

20	01001000	11010100	01111001	11010001
16	11010111	01011010	10000100	00001000
12	01001010	11001010	01000111	01000000
8	00000000	00000000	00000000	00000000
4	00000000	00000000	00000000	00000100
0	01001000	11010100	01111001	11010001

Address

Data

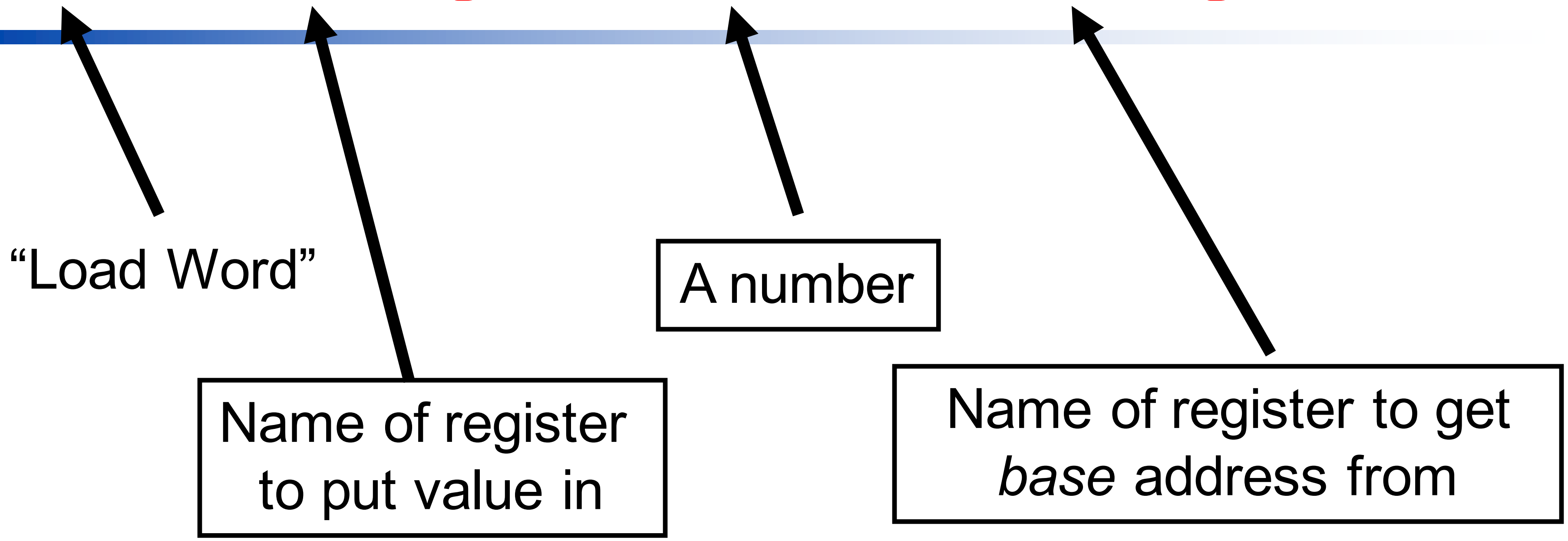
Load Instructions

- *Load* means to move from memory into a register.
- The load instruction needs two things:
  - which register??
  - which memory location (the address)??

`lw`: Load Word

- The *load word* instruction needs to be told an address that is a multiple of 4.
- In MIPS, the way to specify an address is as the sum of:
  - a constant
  - name of a register that holds an address.
  - here we have only 2 register operands and the 3rd is the constant (a compromise!!)

`lw destreg, const(addrreg)`


"Load Word"
Name of register to put value in
A number
Name of register to get *base* address from

address = contents of (***addrreg*** + ***const)***

Example: `lw $s0, 4($s3)`

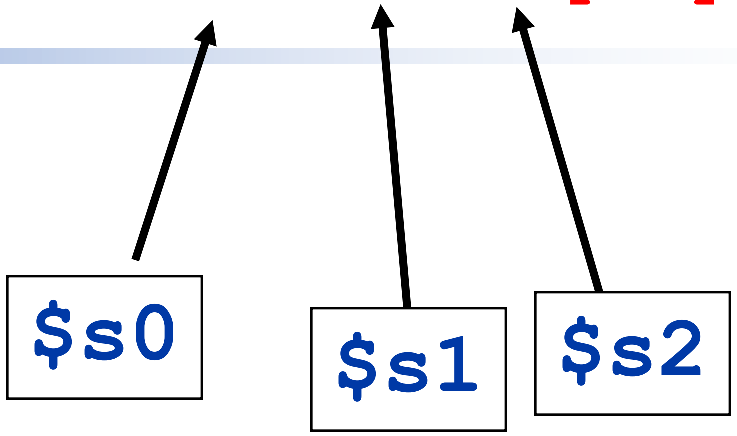
If `$s3` has the value `100`, this will copy the word at memory location `104` to the register `$s0`.

```
$s0 <- Memory[104]
```

Why the weird *address mode?*

- We need to supply a *base* (the contents of the register) and an *offset* (the constant).
- Why not just specify the address as a constant?
  - some instruction sets include this type of addressing.
- It simplifies the instruction set and helps support arrays and structures.

Integer Array Ex: a=b+c[8];



```
lw $t0,8($s2)         # $t0 = c[8]
add $s0, $s1, $t0   # $s0=$s1+$t0
```

Is this right?

Words vs. Bytes

- Each byte in memory has a unique address.
- If the integer array *C* starts at address 100:

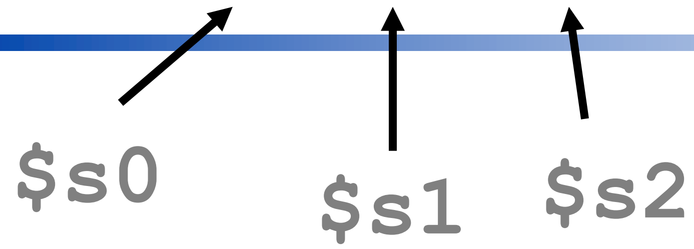
`C[0]` starts at address `100`

`C[1]` starts at address `104`

`C[2]` starts at address `108`

`C[i]` starts at address `100 + i*4`

a=b+c[8]; *(fixed)*



address of **c[8]** is **c+8*4**

```
lw $t0,32($s2)          # $t0 = c[8]
add $s0, $s1, $t0       # $s0=$s1+$t0
```

Moving from Register to Memory

- *Store* means to move from a register to memory.
- The store instruction looks like the load instruction - it needs two things:
  - which register
  - which memory location (the address).

```
sw srcreg, const(addrreg)
```

"Store Word"

Name of register to get value from

A number

Name of register to get *base* address from

Actual address = (*addrreg* + *const*)

Example: `sw $s0, 4($s3)`

If $s3 has the value 100, this will copy the word in register $s0 to memory location 104.

```
Memory[104] <- $s0
```

Example C to MIPS task…

- Write the MIPS instructions that would correspond to the following C code:

```
c[3]=a+c[2];
```

- assume that `a` is `$s0` and that `c` is an array of 32 bit integers whose starting address is in `$s1`

c[3]=a+c[2];

```
lw  $t0, 8($s1)      # $t0 = c[2]
add $t0, $t0, $s0    # $t0=$t0+$s0
sw  $t0, 12($s1)     # c[3] = $t0
```

Variable Array Index: `a=b+c[i]`

- Now the index to the array is a variable.
- We have to remember that the address of **`c[i]`** is the base address + **`4*i`**
- We haven't done multiplication yet, but we can still do this example.

a=b+c[i];

```
add $t0,$s3,$s3      # $t0=i+i
add $t0,$t0,$t0      # $t0=i+i+i+i
add $t0,$t0,$s2      # $t0=c+i*4
lw  $t1,0($t0)       # $t1=c[i]
add $s0,$s1,$t1      # $s0=b+c[i]
```

MIPS Instruction Summary

- MIPS has 32 32-bit registers with names like **`$s0`**, **`$s1`**, **`$t0`**, **`$t1`**, …
- Data must be in registers for arithmetic operations.
- We've seen 2 arithmetic ops: **`add`** & **`sub`**
  - 3 operands – all registers.
- 2 Data transfer instructions: **`lw`**, **`sw`**
  - base/index addressing

MIPS Machine Language

- The processor doesn't *understand* things like this:

```
add $s0,$s0,$s2
```

- It does *understand* things like this:

```
10000101001010001100010000000101
```

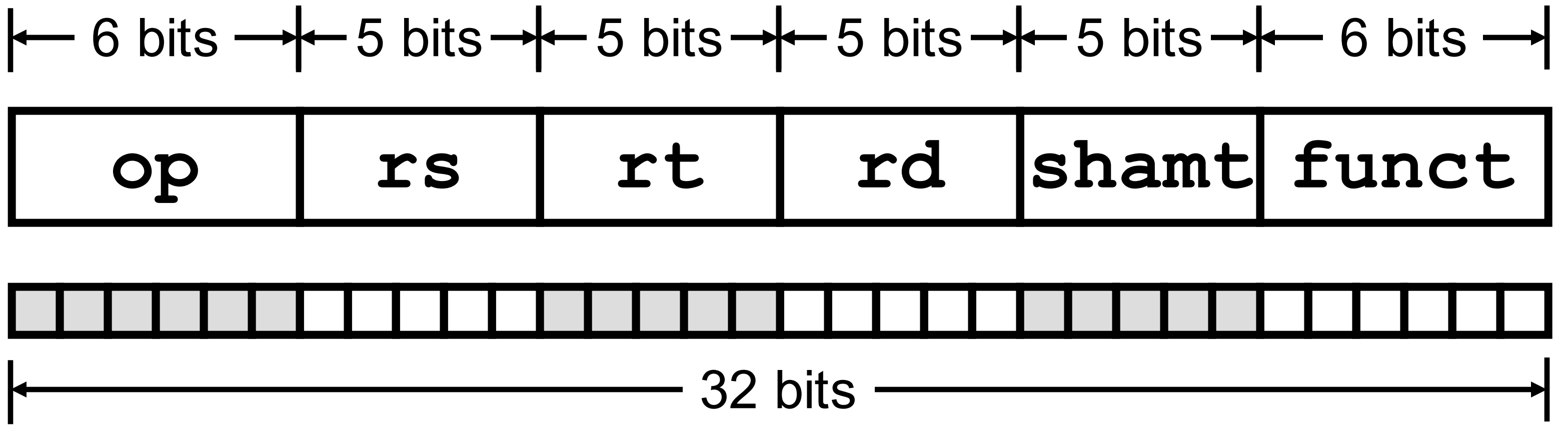
MIPS Machine Code Instructions

- Each instruction is encoded as 32 bits.
  - many processors have *variable* length instructions.
- There are a few different *formats* for MIPS instructions
  - which bits mean what.

Instruction Formats

- break up the 32 bits in to *fields.*
- Each field is an encoding of part of the instruction:
  - fields that specify what registers to use.
  - what operation should be done.
  - constants.

MIPS `add` instruction format



This format is used for many MIPS instructions (not just add). Instructions that use this format are called "*R-Type*" instructions.

`op`	`rs`	`rt`	`rd`	`shamt`	`funct`

`op`:	basic operation (opcode)
`rs`:	first register source operand
`rt`:	second register source operand
`rd`:	destination register
`shamt`:	shift amount (we can ignore for now)
`funct`:	function code: indicates a specific type of operation **`op`**

Encodings

- For `add`:
  - `op` is 0_{10} (000000)
  - `funct` is 32_{10} (100000)
- Register encodings:
  - `$s0` is 16_{10} (10000) , `$s1` is 17_{10}, …
  - `$t0` is 8_{10} (01000) , `$t1` is 9_{10}, …

add $s0, $s1, $t0

000000	10001	01000	10000	00000	100000
op	rs	rt	rd	shamt	funct

In HEX, this add instruction is:

0x02288020

MIPS `sub` Instructions

- Same format as the **`add`** instruction.
- **`op`** is 0_{10} (000000)
- **`funct`** is 34_{10} (100010)

sub $s3, $t1, $s0

000000	01001	10000	10011	00000	100010
op	rs	rt	rd	shamt	funct

In HEX, this sub instruction is:

0x01309822

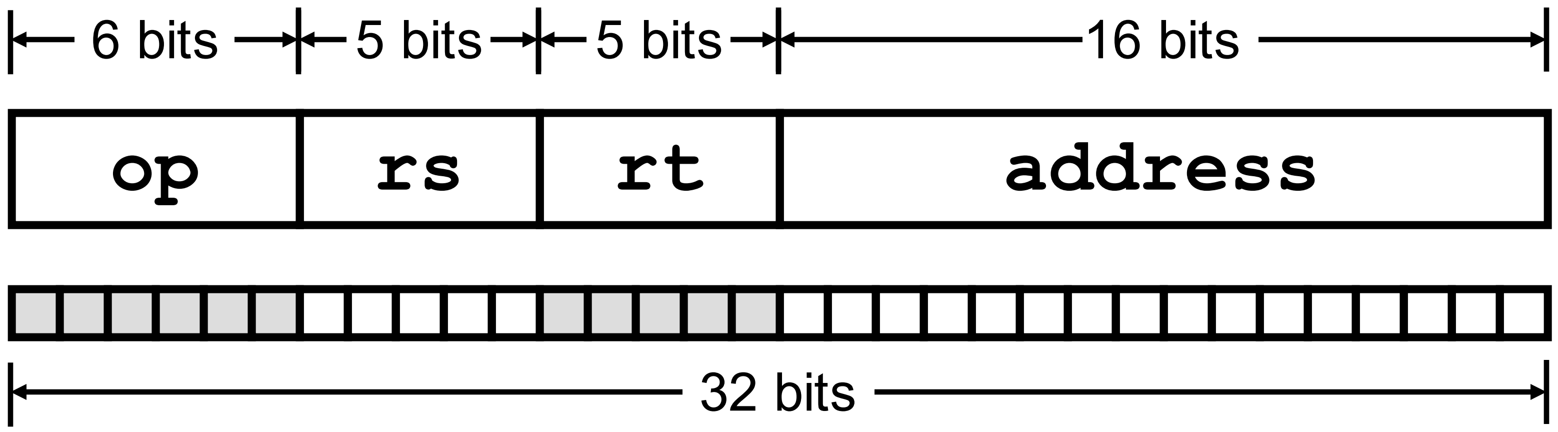
LW/SW Instruction Format?

- Different format is necessary (no place to put the constant)
- What do we do with the constant (index)? It is a 16 bit number?
- Should we KLUDGE fields together from the "*R-type*" format??
- This brings up the 3rd design principal:

  GOOD DESIGN DEMANDS GOOD COMPROMISE!!!

- Create a new instruction format:"*I-Type*"

MIPS I-Type instruction format



`rs` is the base register

`rt` is the destination of a load (source of a store)

`address` is a *signed* integer

`lw` and `sw` instructions

`lw`: The op field is 35_{10} (100011)

`sw`: The op field is 43_{10} (101011)

Only 1 bit difference!

lw $s0, 24($t1)

100011	01001	10000	0000000000011000
op	rs	rt	address

sw $s0, 24($t1)

101011	01001	10000	0000000000011000
op	rs	rt	address

Sample Exercise

- What is the MIPS machine code for the following C statement:

```
c[3] = a + c[2];
```

c[3] = a + c[2];

- First we can work on the Assembly instructions - assume a is $s0 and the base address of c is in $s1:

```
lw  $t0, 8($s1)  # $t0 = c[2]
add $t0,$t0,$s0 # $t0 = a+c[2]
sw  $t0, 12($s1) # c[3] = a+c[2]
```

lw $t0, 8($s1)

100011	10001	01000	0000000000001000
op	rs	rt	address

op is 35_{10} for **lw**

rs is 17_{10} for **$s1**

rt is 8_{10} for **$t0**

address is 8_{10}

add $t0,$t0,$s0

000000	01000	10000	01000	00000	100000
op	rs	rt	rd	shamt	funct

op is 0_{10} for **add**

funct is is 32_{10} for **add**

rs is 8_{10} for **$t0**

rt is 16_{10} for **$s0**

rd is 8_{10} for **$t0**

shamt is 0_{10}

`sw $t0, 12($s1)`

`101011`	`10001`	`01000`	`0000000000001100`
`op`	**`rs`**	**`rt`**	**`address`**

`op` is 43_{10} for **`sw`**

`rs` is 17_{10} for **`$s1`**

`rt` is 8_{10} for **`$t0`**

`address` is 12_{10}

Machine code for `c[3] = a+c[2];`

```
10001110001010000000000000001000     lw $t0, 8($s1)
00000001000100000100000000100000     add $t0,$t0,$s0
10101110001010000000000000001000     sw $t0, 12($s1)
```

Congratulations – you are now on your way to being qualified to be an *assembler*!

MIPS Instructions (so far)

- We've seen 2 arithmetic ops: **`add`** & **`sub`**
  - 3 operands - all registers.
- 2 Data transfer instructions: **`lw`**, **`sw`**
  - base/index addressing
- Two machine language instruction formats:
  - R-Type (3 registers)
  - I-Type (2 registers and offset)

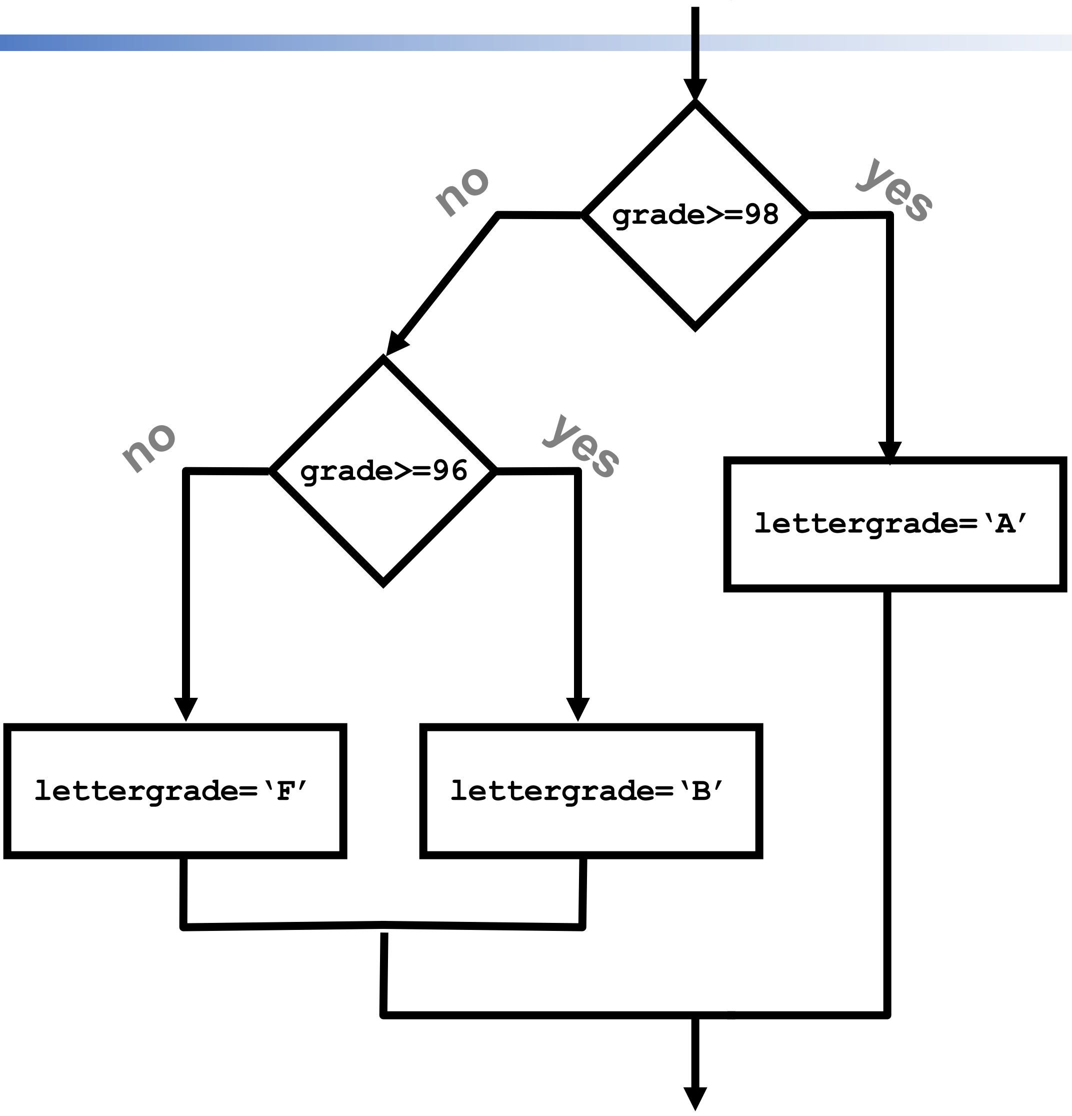
Jumping Around

- There are instructions that change the sequence of instructions fed to the processor, that *jump* to a new part of the program.
- Jumping is also called *branching.*
- Sometimes we want to jump only when some condition is true (or false).

C Program that requires jumping

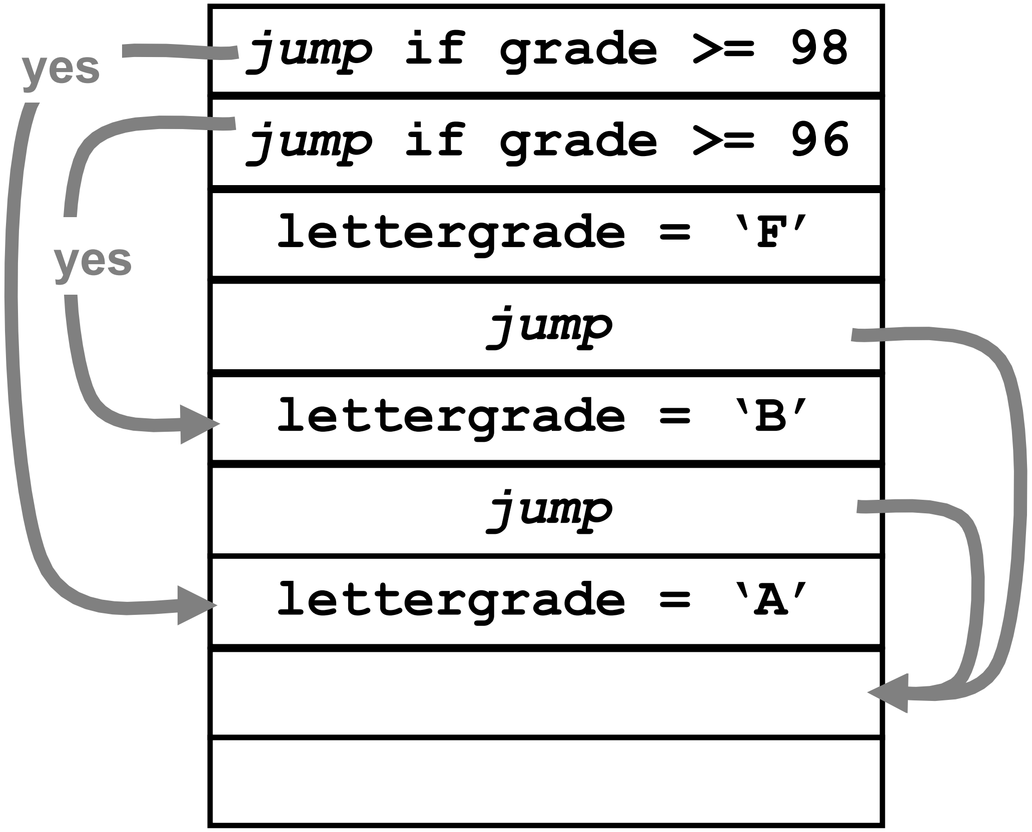
```
if (grade >= 98)
    lettergrade = ‘A’;
else if (grade >= 96)
    lettergrade = ‘B’;
else
    lettergrade = ‘F’;
```

The flow of the program



Possible layout in memory

```
if (grade >= 98)
    lettergrade = 'A';
else if (grade >= 96)
    lettergrade = 'B';
else
    lettergrade = 'F';
```



MIPS instructions for jumping

```
beq reg1, reg2, address
```

Branch if Equal : if the contents of register ***reg1*** are equal to the contents of register ***reg2*** then jump to ***address***.

If the registers are not equal, don't do anything special (continue on to the next instruction).

`bne reg1, reg2, address`

Branch if Not Equal: if the contents of register ***`reg1`*** is not equal to the contents of register ***`reg2`***, then jump to ***`address.`***

If the registers are equal, don't do anything special (continue on to the next instruction).

`beq r1,r2,address`: What is *address*?

- In assembly language we create *labels* in the program that can be used as the address (example next slide).

- In machine language the address is an offset from the location of the current instruction.

Example:

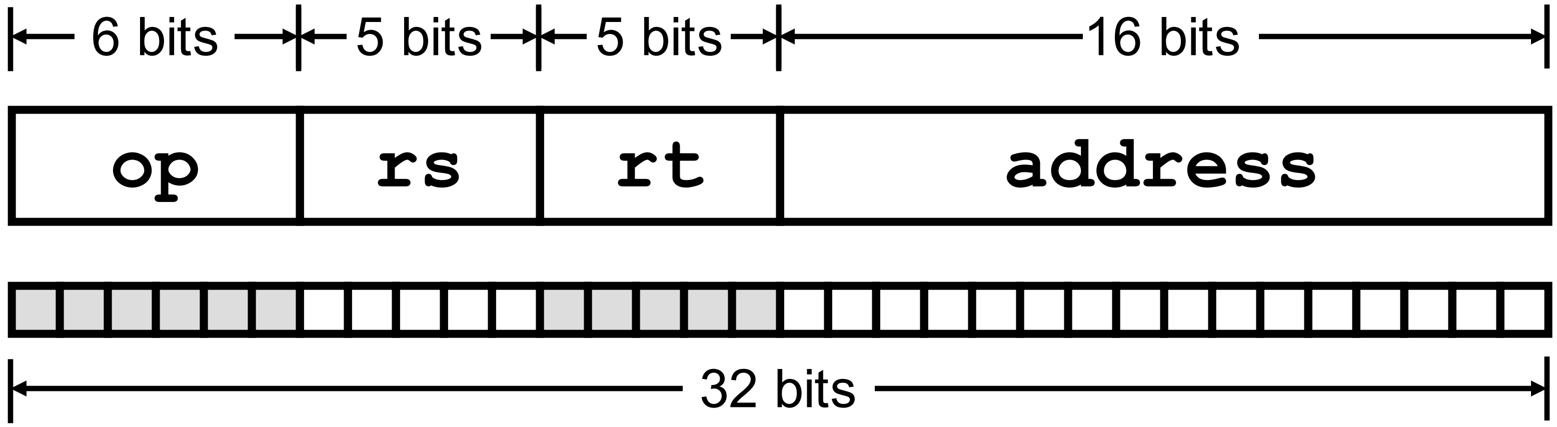
```
        $s0   $s1  $s2    $s3
if   (a==b)   c=c+d;
a=b+b;
```

```
       bne   $s0,$s1,L1       # go to L1 if a!=b
       add   $s2,$s2,$s3      # c=c+d
L1:    add   $s0,$s1,$s1      # a=b+b
```

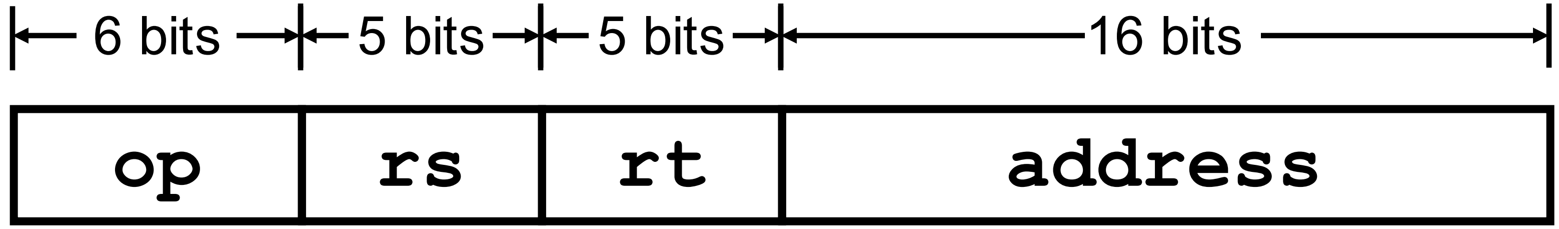
label

Machine Code for `bne` and `beq`

Instruction Format: *I-Type*:



The `address` Field of I-Type Instructions



MIPS supports 32 bit addresses.

If we only have a 16 bit address we can't have very large programs!

The **`address`** field is *relative to the address of the current instruction.*

- recall, all MIPS instructions are 32 bits…this will be an example of our 4th design principal…but we'll get to that later.
- this is just base/index addressing with the PC as the base register. So, what the heck is a PC???

`PC`: The Program Counter

- There is a special register called the *program counter* that holds the address of the current instruction.
- Normally, this register is incremented by 4 each instruction (MIPS instructions are 4 bytes each).
- When a branch happens - the **`address`** is added to the **`PC`** register.

The value of `PC` during an instruction.

- During the *execution* of an instruction, the processor <u>always</u> adds 4 to the **`PC`** register.
- This happens *very early* in the instruction.
- As far as we are concerned the *PC* always holds the address of the *next* instruction.

Instruction Alignment

- Since MIPS instructions are always stored in memory at an address on a *word boundary* (divisible by 4), the offset specified is actually a *word* offset (not a byte offset).

- A value of 1 means "add 4 to PC".
- A value of 100 means "add 400 to PC".

Assembly vs. Machine Code

```
      bne   $s0,$s1,L1      # go to L1 if a!=b
      add   $s2,$s2,$s3     # c=c+d
L1:   add   $s0,$s1,$s1     # a=b+b
```

- The assembler calculates the difference between the address of the instruction following the **bne** instruction and the instruction labeled L1.
- This difference is used in the address field of the machine code for the **bne** instruction.
- In this case the difference is 1 (1 instruction).

`bne, beq` limitations

- The offset from the PC is actually a *signed* integer value.
  - we can jump backwards or forwards.
- The maximum offset is:

  2^{15} instructions = 2^{17} bytes

- The MIPS memory is 2^{32} bytes !

Unconditional Jump

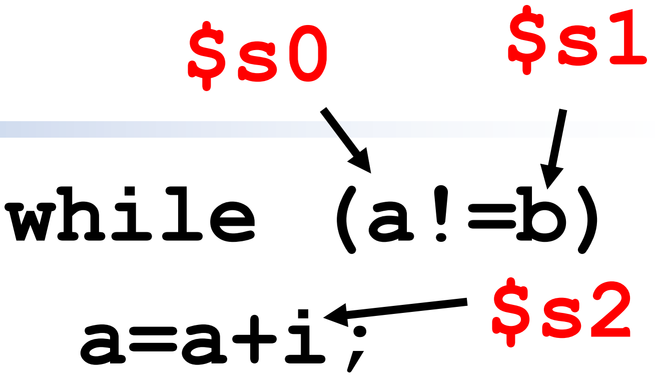
- MIPS includes an instruction that always jumps:

```
j address
```

- In assembly language we just use a label again.

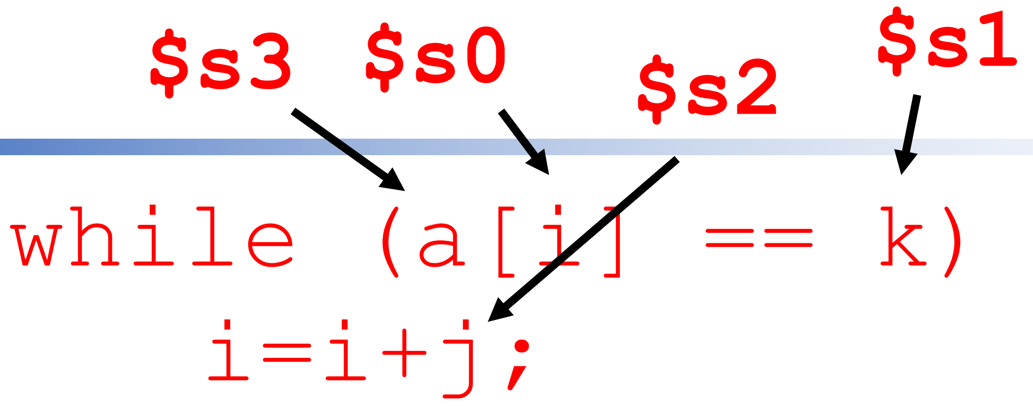
```
j L1
```

Loops in Assembly



```
while (a!=b)
    a=a+i;
```

```
Loop:   beq $s0,$s1,Eol
        add $s0,$s0,$s2
        j   Loop
Eol:
```

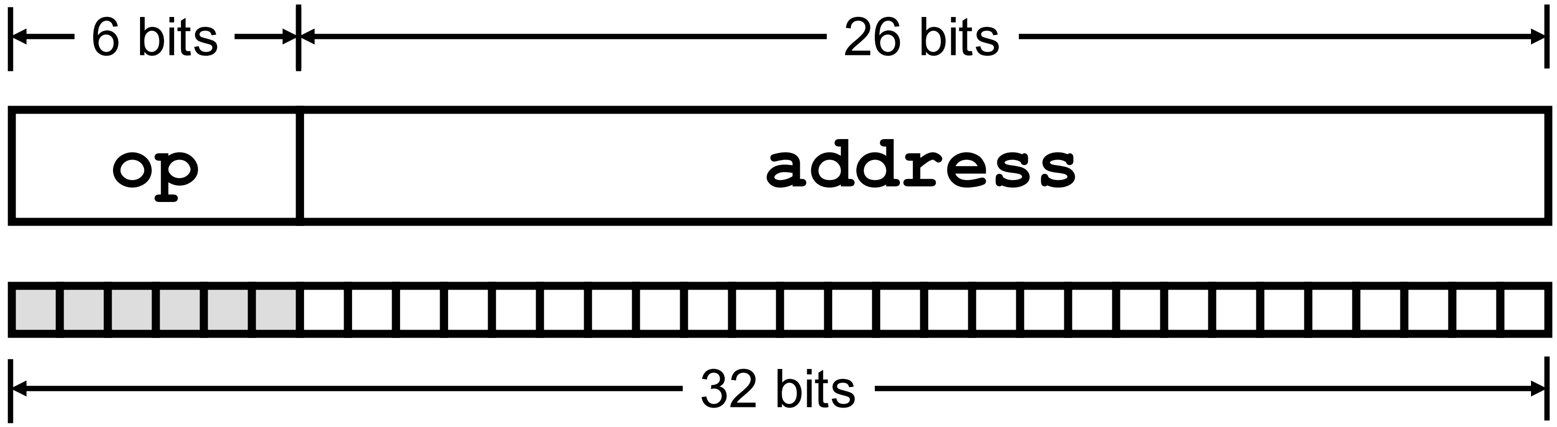


```
while (a[i] == k)
      i=i+j;
```

```
L1:   add   $t0,$s0,$s0      # $t0=i*2
      add   $t0,$t0,$t0      # $t0=i*4
      add   $t0,$t0,$s3      # $t0 = addr of a[i]
      lw    $t1,0($t0) # $t1 = a[i]
      bne   $t1,$s1,L2 # if a[i]!=k goto L2
      add   $s0,$s0,$s2      # i=i+j
      j          L1          # goto L1
L2:
```

j instruction format

New Instruction Format: *J-Type*:



J-Type address field

- The address field is treated as an instruction address (not a byte address).
- The *rightmost* 28 bits of the PC are replaced with this address (which is left-shifted 2 bits).
- It's **not** relative to the PC!
- We have to build very large programs (with more than 2^{26} instructions) very carefully!

↑

Actually the compiler/assembler/linker takes care of this for us!

What about < and > ?

- We often want to compare numbers and jump if one number is less-than or greater than another number.
- MIPS does not include a conditional jump instruction that does this, instead there are some instructions that compare numbers and *store the result in a register.*
- We can then use **`beq`**, **`bne`** with the result of the comparison.

Set if Less Than: `slt`

`slt` ***`dstreg, reg1, reg2`***

`dstreg` is set to a 1 if `reg1` is less than `reg2`

`dstreg` is set to a 0 if `reg1` is not less than `reg2`

`if (a<b)`

$s0

$s2

```
slt $t0,$s0,$s2  # $t0 <- a<b
bne $t0,$zero,L1 # jump if a<b
```

`$zero` is a MIPS register that always holds the value 0!

Another unconditional jump

`jr reg`

- "Jump Register"
  - put the contents of the register in to the PC register.
- The book describes using this with a *jump table* to build a C switch statement.

Instruction Summary (so far)

- Arithmetic: **`add sub`**
- Data Movement: **`lw sw`**
- Jumping around: **`bne beq slt j jr`**
- We can almost build real programs…!!

Byte operations

- In C programs we often deal with strings of characters (ASCII characters).
- Each character is 1 byte.
- We need instructions that can deal with 1 byte at a time.

Load Byte: `lb`

`lb` ***`destreg, const(addrreg)`***

- Moves a single byte from memory to the rightmost 8 bits of **`destreg`**.
- The other 24 bits of **`destreg`** are set to 0
  - actually the byte is *sign extended* (more on this when we talk about arithmetic).
- Base/Index addressing (just like **`lw`**).

Store Byte: sb

sb ***srcreg, const(addrreg)***

- Moves a single byte from the rightmost 8 bits of **destreg** to memory.
- Base/Index addressing (just like **sw**).

Copying a C string.

- C strings are terminated with a 0.
  - the last byte in the string has the value $00000000_2 = 0_{10}$
- Strings are like arrays of characters.
  - now each array item is 1 byte only!

Assembly for `strcpy(str1,str2)`

dest source

- We aren't yet worried about making this a real subroutine, we just want the code that can do the copying.
- Assume register `$s1` holds the address of `str1`, and `$s2` holds the address of `str2`
- We need a loop that copies from the address in `$s2` to the address in `$s1`
  - `increments $s2 and $s1 each time.`

A start at strcpy

```
Loop:lb    $t0,0($s2) # $to = *str2
           sb    $t0,0($s1) # *str1 = $t0
```

need to increment $s1,$s2

```
           bne   $t0,$zero,Loop  #
```

Incrementing a register

- We could assume some register has the value 1 in it.
  - it would have to get there some how!
- New instruction: Add *Immediate*

What about constants?

- Should we read them from memory??
  - What performance problems does this create?
  - It turns out, instructions with constants are very frequent...
  - What does Amdahl's Law say??
- ***Principal #4: Execute the Common Case Fast***
  - here, put constants directly into the instruction!!

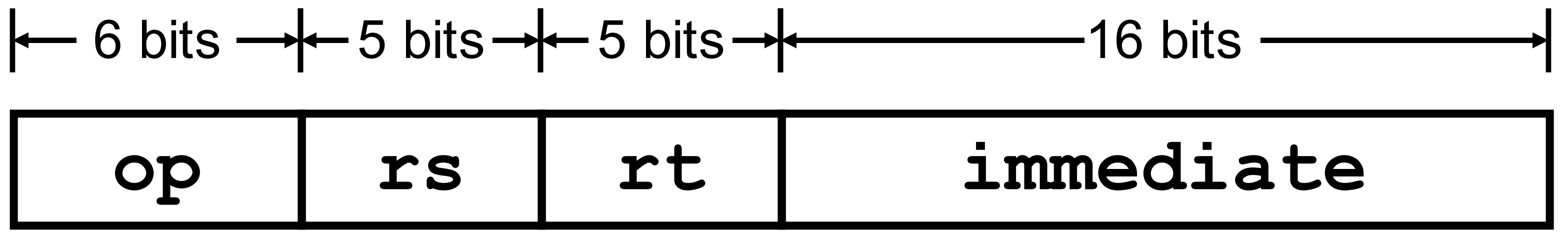
Add Immediate

addi ***destreg, reg1, const***

Adds a constant to **reg1** and puts the sum in **destreg**.

The term "immediate" means the value (the constant) is already available to the processor (it's part of the instruction).

addi *is an I-Type instruction*



rt is the destination register

rs is a source operand.

immediate is the constant.

16 bit "signed" constant!

Incrementing a register

- To add 1 to the register $s0:

```
addi    $s0,$s0,1
```

- To add 1234 to the register $t3:

```
addi    $t3,$t3,1234
```

- To add 1,000,000 to the register $s2: ???

Finishing strcpy

```
Loop:lb    $t0,0($s2)# $to = *str2
           sb    $t0,0($s1)# *str1 = $t0
           addi  $s2,$s2,1        # str2++
           addi  $s1,$s1,1        # str1++
           bne   $t0,$zero,Loop  #
```

32 bit constants

- Sometimes we need to deal with 32 bit constants!
  - not often, but it happens…
- We can now load the lower 16 bits with any constant value:

```
    addi    $s0,$zero,const
```

- We need some way to put 16 bits in to the left half of a register.

Load Upper Immediate: `lui`

`lui` *`destreg, const`*

- `const` is a 16 bit immediate value.
- The lower 16 bits of `destreg` are all set to 0! (have to load the upper half first!)

Immediates are fun!

- There is also a version of `slt` that uses an immediate value:

  `slti` ***`destreg, reg1,const`***

- Will set ***`destreg`*** to 1 if ***`reg1`*** is ***less than*** the 16 bit constant.

Write this in MIPS Assembly

```
for (i=0;i<10;i++) {
  a[i] = a[i+1];
}
```

`a` is an array of `char`!

Solution: the address of `a` is in `$s1`

```
      add  $s0,$zero,$zero # i=0
L1:   slti $t1,$s0,10      # i<10?
      beq  $t1,$zero,L2         #   no-jump
      add  $t2,$s0,$s1     # t2 is addr of a[i]
      lb   $t3,0($t2) # t3 is a[i]
      addi $t2,$t2,1       # t2 is addr of a[i+1]
      sb   $t3,0($t2) # a[i+1] = t3
      addi $s0,$s0,1       # i++
      j    L1              # go to L1
L2:
```

SPIM

Ref: Appendix A, Web Links

http://pages.cs.wisc.edu/~larus/spim.html

MIPS Simulation

- SPIM is a simulator
  - reads a MIPS assembly language program.
  - simulates each instruction.
  - displays values of registers and memory
  - supports breakpoints and single stepping
  - provides simple I/O for interacting with user.

SPIM Versions

- SPIM is the command line version.
- XSPIM is X-Windows version (Unix workstations).
- There is also a Windows version.

SPIM Program

- MIPS assembly language.
- Must include a label "main" – this will be called by the SPIM startup code (allows you to have command line arguments).
- Can include named memory locations, constants and string literals in a "data segment".

General Layout

- Data definitions start with `.data` directive
- Code definition starts with `.text` directive
  - "text" is the traditional name for the memory that holds a program.
- Usually have a bunch of subroutine definitions and a "main".

Simple Example

```
        .data           # data memory
foo     .word 0         # 32 bit variable

        .text           # program memory
        .align 2        # word alignment
        .globl main     # main is global

main:
```

Data definitions

- You can define variables/constants with:
  - **.word** : defines 32 bit quantities.
  - **.byte**: defines 8 bit quantities
  - **.asciiz**: zero-delimited ascii strings
  - **.space**: allocate some bytes

Data Examples

```
            .data
prompt:     .asciiz “Hi Dr. C”
msg:        .asciiz  “The answer is ”
x:          .word    0
y:          .word    0
str:        .space 100
```

Simple I/O

- SPIM provides some simple I/O using the "syscall" instruction. The specific I/O done depends on some registers.
  - You set `$v0` to indicate the operation.
  - Parameters in `$a0`, `$a1`

I/O Functions

$v0	Function	Parameter
1	`print_int`	`$a0` is int
4	`print_string`	`$a0` is address of string
5	`read_int`	returned in `$v0`
8	`read_string`	`$a0` is address of buffer, `$a1` is length

Example: Reading an int

```
        addi  $v0,$zero,5
        syscall

  # now $a0 has the integer typed by
  # a human in the SPIM console
```

Printing a string

```
            .data
msg:        .asciiz  "SPIM IS FUN"

main:       li $v0,4
            la $a0,msg
            syscall
```

pseudoinstruction: load immediate

pseudoinstruction: load address

SPIM subroutines

- The stack is set up for you - just use `$sp`
- You can view the stack in the data window.
- `main` is called as a subroutine (have it return using `jr $ra`).
- We'll talk a great deal about subroutines

Sample SPIM programs (on the web)

- `multiply.s`: multiplication subroutine based on repeated addition and a test program that calls it.
- `fact.s`: computes factorials using the multiply subroutine.
- `sort.s`: the sorting program from the text.
- `strcpy.s`: the strcpy subroutine and test code.

MIPS Subroutines and Programs

Ref: Chapter 2

Subroutines

```
main:
            # multiply 3 x 2
            addi $a0,$zero,3
            addi $a1,$zero,2

            # call the subroutine
            jal multiply

            # print out the result
            move $s0,$v0
            li $v0,4
            la $a0, msg
            syscall

            li $v0,1
            move $a0,$s0
            syscall

            li $v0,10
            syscall
```

```
# mult subroutine needs some registers
# so we save $t0 first
# 2 arguments $a0 and $a1 are multiplied
# using repeated addition

multiply:
            sub $sp,$sp,4   # make room for $t0
            sw  $t0,0($sp)  # put t0 on the stack

            # start with $t0 = 0
            add $t0,$zero,$zero
mult_loop:
            # loop on a1
            beq $a1,$zero,mult_eol

            # add another $a0
            add $t0,$t0,$a0
            # decrement $a1
            sub $a1,$a1,1
            j mult_loop

mult_eol:
            # put the result in $v0
            add $v0,$t0,$zero

            # restore $t0
            lw $t0,0($sp)
            add $sp,$sp,4
            # return to caller
            jr $ra
```

Subroutine Issues

- How to call a subroutine
  - how to pass parameters
  - how to get the return value
- How to write a subroutine
  - where to look for parameters
  - saving registers
  - returning a value
  - returning to the *caller*

Special Registers

$a0-$a4: argument registers

- this is where we put arguments before calling a subroutine.

$v0, **$v1**: return value registers

- where subroutines put return values

$ra: return address register

- holds the address the subroutine should jump to when it's done.

Jump and Link: `jal address`

- Puts the address of the next instruction (**`PC+4`**) in the **`$ra`** register.
- Jumps to the specific address.
- Addressing mode is just like the `j` instruction (26 bit absolute address).

Returning from the Subroutine

- Assuming the subroutine doesn't clobber the `$ra` register:
  - when the subroutine is done, it jumps to the address in `$ra`

```
jr $ra
```

What if the subroutine uses a register?

Accepted convention:

`$t0, $t1, … $t7` are always OK to use.

- if you call a subroutine and you need the value of `$t0` to be the same after the call, you must save it in memory!
- *caller saves* `$t0` … `$t7`

`$s0-$s7` must not be changed by a subroutine unless you save them first!

- If you need them in your subroutine you need to save the previous value and restore them before returning.
- *callee saves* `$s0` … `$s7`

Saving registers and the Stack

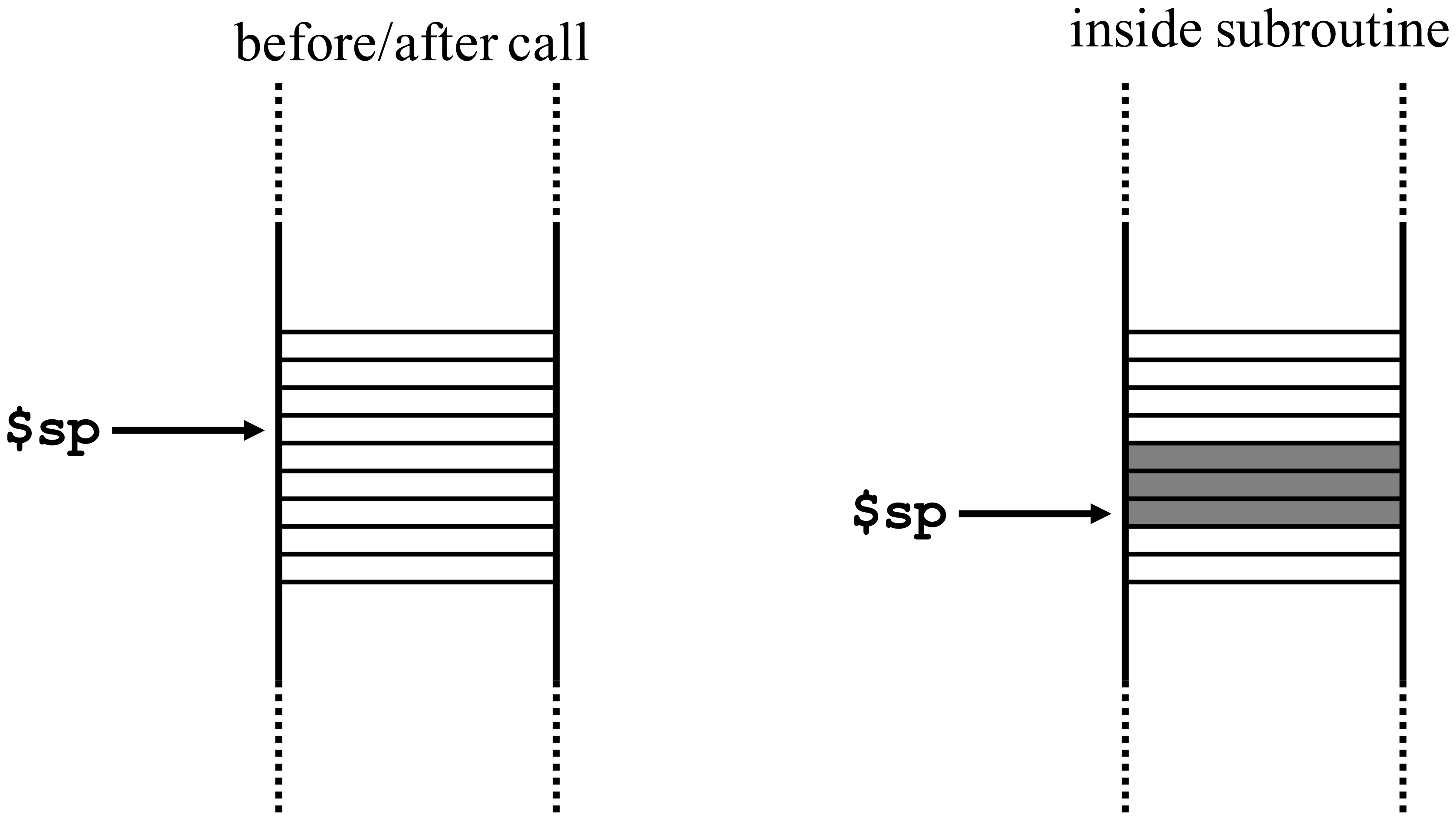
- Most of the time we use whatever registers we want inside subroutines.
  - must save and restore `$s0-$s7`
- This happens so often there is a special register and data structure used to support saving and restoring registers.

The Stack!!!!!

The Stack

- The stack is an area of memory reserved for the purpose of saving registers.
- The `$sp` register (*stack pointer*) holds the address of the *top* of the stack.
- The stack grows and shrinks as registers are saved and restored.

$sp and Memory



Stack handling code

- Suppose your subroutine needs to use 3 registers: `$s0`, `$s1` and `$s2`:
  - first make room for saving three words by subtracting 12 from the stack pointer

```
sub $sp,$sp,12
```

  - now put copies of the three registers on the stack.

```
sw $s0,0($sp)
sw $s1,4($sp)
sw $s2,8($sp)
```

Stack handling code (cont.)

- Before returning, your subroutine should restore the 3 registers:

```
lw $s2,8($sp)
lw $s1,4($sp)
lw $s0,0($sp)
```

- And put the stack pointer back to its original value:

```
add $sp,$sp,12
```

Why bother?

- We write subroutines so that they can be called from *any other code*.
  - as far as the *caller* is concerned, `$s0-$s7` don't change.
- The stack provides a single mechanism that will work no matter who called the subroutine.

Exercise

- Create a multiplication subroutine.
  - $a0 is multiplied by $a1 and the product is returned in $v0

- We've already looked at the multiply code, all we need to do is make this a subroutine.

multiply in 'C'

```
int multiply(int x, int y) {
   prod=0;
   while (y>0) {
     prod = prod + x;
     y--;
   }
   return(prod);
}
```

Assembly Multiply

```
int multiply(int x, int y) {
     prod=0;
     while (y>0) {
        prod = prod + x;
        y--;
     }
     return(prod);
}
```

```
multiply:
        add $t0,$zero,$zero  # prod=0
m_loop:
        beq $a1,$zero,m_eol  # while y>0
        add $t0,$t0,$a0# prod = prod+x
        subi $a1,$a1,1       # y--;
        j m_loop
m_eol:
        add $v0,$t0,$zero    # return(prod)
        jr $ra
```

Not a typical example!

- `multiply` doesn't need many registers and it doesn't call any subroutines.
  - no need to save and restore registers
- Let's go back and make our assembly version of strcpy a subroutine.

strcpy in C

```
strcpy( char *str1, char *str2 )
{
  do
  {
      *str1 = *str2;
      str1++;
      str2++;
  }
  while( *str1 );
}
```

strcpy in Assembly:

`str1` is `$s1` and `str2` is `$s2`

```
Loop:lb    $t0,0($s2) # $to = *str2
           sb    $t0,0($s1) # *str1 = $t0
           addi  $s2,$s2,1         # str2++
           addi  $s1,$s1,1         # str1++
           bne   $t0,$zero,Loop  #
```

- Uses registers `$s0`, `$s1` and `$t0`
- Remember our convention: callee (the subroutine) must save and restore `$s0-$s7`

strcpy subroutine

```
strcpy:
       addi $sp,$sp,-8      # make room for 2 regs
       sw $s2,4($sp)        # save $s2
       sw $s1,0($sp)        # save $s1
       add $s1,$a0,$zero    # move $a0 to $s1
       add $s2,$a1,$zero    # move $a1 to $s2
Loop:
       lb     $t0,0($s2)           # $to = *str2
       sb     $t0,0($s1)           # *str1 = $t0
       addi   $s2,$s2,1            # str2++
       addi   $s1,$s1,1            # str1++
       bne    $t0,$zero,Loop       # jump if not done

       lw $s1,0($sp)        # restore $s0
       lw $s2,4($sp)        # restore $s1
       addi $sp,$sp,8       # adjust stack
       jr $ra               # return
```

Recursive Exercise

Write the MIPS Assembly Language code for the following C program:

```
int factorial( int x ) {
  if (x<1) return 1;
  else return x * factorial(x-1);
}
```

Recursion - Issues

- Since this subroutine calls another subroutine (in this case it calls itself!):
  - we need to save `$ra`
  - we need to save any temp registers (`$t0-$t7`) before calling a subroutine.
    - only if we need the value of a temp register to still be the same after the call!

Outline of `factorial` subroutine

- save registers `$ra`, and `$a0` (the argument `x`)
- check to see if `x<1`, if so just return 1
- if `x>=1`:
  - call `factorial(x-1)` and put result in `$a1`
  - put `x` in `$a0`
  - call multiply: result in `$v0`
  - restore `$ra` and `$a0`
  - return

factorial (part 1)

```
factorial:
     # make room for 2 registers
     addi $sp,$sp,-8

     # save $ra and $a0 on stack
     sw $a0,4($sp)
     sw $ra,0($sp)

     slti $t0,$a0,1        # is x < 1 ?
     bne $t0,$zero,L1      # yes - go to L1
```

factorial (part 2) when x>=1

```
sub $a0,$a0,1          # x--;
jal factorial          # call fact(x-1)

# Now multiply the result by x
# a0 is no longer x,
#     but we still have it on the stack

lw $a0,4($sp)
add $a1,$v0,$zero      # $v0 is fact(x-1)
jal multiply           # get the product
```

factorial (part 3)

```
# restore $a0 and $ra before returning
# multiply may have changed $a0
#    (so we must restore again)
# $v0 is already the return value

  lw $ra,0($sp)     # restore $ra
  lw $a0,4($sp)     # restore $a0
  add $sp,$sp,8     # restore the stack
  jr $ra
```

factorial **(part 4) x<1**

```
L1:
   # x<1 so we just return 1
   addi $v0,$zero,1
   # $a0 and $ra have not changed,
   # so there is no need to restore
   # but we need to restore the stack
   add $sp,$sp,8
   jr $ra
```

Ex: Simulate `factorial(3)`

- Step through the code, keeping track of:
  - all the used registers
  - $sp and the contents of the stack
- Spim makes this easy! We'll talk about this shortly…

What about saving `$t0-$t7`?

- The convention says we should expect subroutines to use `$t0-$t7`.
- If we use them and need the value to be the same after a subroutine call – we need to save them before calling the subroutine.
- We also need to restore them after calling the subroutine.

Saving registers

- The code is the same - use the stack:

```
add $sp,$sp,-8
sw $ra,0($sp)
sw $t0,4($sp)

jal whatever

lw $ra,0($sp)
lw $t0,4($sp)
add $sp,$sp,4
```

```
add $sp,$sp,-12
sw $t1,8($sp)
sw $t0,4($sp)
sw $ra,0($sp)

jal whatever

lw $t1,8($sp)
lw $t0,4($sp)
lw $ra,0($sp)
add $sp,$sp,4
```

Writing & Calling Subroutines

- When calling a subroutine:
  - you don't need to worry about `$s0-$s7`, they won't change.
  - you do need to worry about `$t0-$t7` they may change.
- When writing a subroutine:
  - you need to save/restore the callers `$s0-$s7` if you use them.
  - `$t0-$t7` are always free to use

More Writing Subroutines

- Careful with `$ra` - if you call a subroutine this will change `$ra` (and your return won't work!). Might need to save/restore `$ra`.
- Careful with `$a0-$a4` and `$v0-$v1`.
- ALWAYS: Make sure `$sp` is the same as when you were called!!!!

Pseudoinstructions

- There are many instructions you can use in MIPS assembly language that don't really exist!
- They are a *convienence* for the programmer (or compiler) – just a shorthand notation for specifying some operation(s).

MIPS `move` pseudoinstruction

`move` *`destreg, sourcereg`*

There is no move instruction, but the assembler lets us pretend.

The assembler can achieve this using `add` and `$zero`:

`move $s0, $s1` is really `add $s0,$s1,$zero`

`Blt`: **Branch if Less Than**

- *Branch if less than* is a pseudoinstruction based on slt and bne:

```
blt $s0,$s1,foo
```

is really:

```
        slt $at,$s0,$s1
        bne $at,foo
```

Register `$at` is reserved for use by the assembler (we can't use it - the assembler needs it for pseudoinstructions).

Some useful pseudoinstructions

`li`	load immediate
`la`	load address
`sgt, sle, sge …`	set if greater than, …
`bge, bgt, ble, blt`	conditional branching