3. More MIPS Instructions

Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions based on input data or values obtained during the computation

- In high-level programming languages, decision-making instruction:
  - **if** statement

- In MIPS, decision-making instructions (or **conditional branches**):
  - `beq` ('branch if equal'):
    - e.g. `beq reg1, reg2, L1`
    - go to statement labeled `L1` if `reg1` and `reg2` have the same value
  - `bne` ('branch if not equal'):
    - e.g. `bne reg1, reg2, L1`
    - go to statement labeled `L1` if `reg1` and `reg2` do not have the same value

Example

- In the following C code segment, `f`, `g`, `h`, `i`, and `j` are variables:

```
        if (i == j) goto L1;
        f = g + h;
   L1:  f = f - i;
```

Assuming that the five variables `f` through `j` correspond to five registers `$s0` through `$s4`, what is the compiled MIPS code?

- **Answer**:

```
      beq $s3, $s4, L1     # go to L1 if i==j
      add $s0, $s1, $s2    # f = g + h (skipped if i==j)
L1:   sub $s0, $s0, $s3    # f = f - i (always executed)
```

Notes:

`L1` corresponds to the address of the `sub` instruction.

More on Branches

- Compilers frequently create branches and labels where they do not appear in the programming language.

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

this code is another way to implement the previous example without using label `L1`

- Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is one of the reasons why coding is faster at that level.

- Besides conditional branches, we also have **unconditional jumps**:
  - `j` ('jump'):
    - e.g. `j L1`
    - always go to statement labeled `L1`

Example

- Assume, as before, that the five variables `f` through `j` correspond to registers `$s0` through `$s4`. What is the compiled MIPS code for this?

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

- **Answer**:

```
        bne $s3, $s4, ElseIf     #if(i!=j) goto Elseif
        add $s0, $s1, $s2
        j   Exit
ElseIf:bne $s3, $s1, Else        #if(i!=j) goto Else
        sub $s0, $s1, $s2
        j   Exit
Else:   add $s0, $s1, $s4
Exit:
```

Example (Cont'd)

- Another Solution:

```
                    beq $s3, $s4, if_match
                    beq $s3, $s1, elseif_match
                    j   else_match
if_match:           add $s0, $s1, $s2
                    j   exit
elseif_match:       sub $s0, $s1, $s2
                    j   exit
else_match:         add $s0, $s1, $s4
exit:
```

- Although this solution is longer, it is more similar to C++ version & looks closer to a switch-case statement
- Could be easier to debug if you need to check for more conditions

Loops

- Decisions are important both for
  - choosing between two alternatives–found in **if** statement
  - iterating a computation–found in **loops**
- In **loops**, decisions are needed to determine when to stop looping
- Commonly used loop constructs in high-level programming languages
  - **while**
  - **for**

Example

- Here is a traditional loop in C:

```
while (save[i] == k)    i += 1;
```

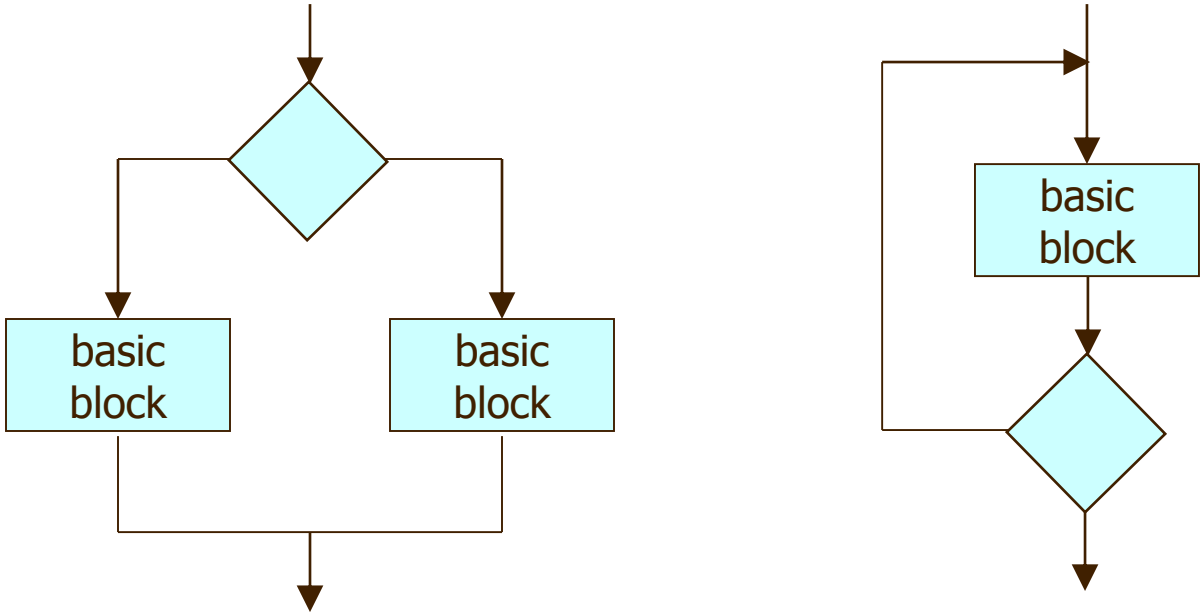
Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array save is in `$s6`. What is the MIPS assembly code corresponding to this C Segment?

- **Answer**:

```
Loop: sll $t1, $s3, 2      # Temp reg $t1 = 4 * i
      add $t1, $t1, $s6    # $t1 = address of save[i]
      lw  $t0, 0($t1)      # Temp reg $t0 = save[i]
      bne $t0, $s5, Exit   # go to Exit if save[i] != k
      addi $s3, $s3, 1     # i = i + 1
      j   Loop
Exit:
```

Basic Blocks

- A **basic block** is a sequence of instructions without branches and branch targets (except possibly at the end and at the beginning)



- One of the first early phases of compilation is breaking the program into basic blocks.

Example of Basic Blocks

- #1

```
Loop:  sll $t1, $s3, 2
       add $t1, $t1, $s6
       lw  $t0, 0($t1)
       bne $t0, $s5, Exit
       add $s3, $s3, 1
       j   Loop
Exit:
```

- #2

```
       bne $s3, $s4, ElsIf
       add $s0, $s1, $s2
       j   Exit
ElsIf: bne $s3, $s1, Else
       sub $s0, $s1, $s2
       j   Exit
Else:  add $s0, $s1, $s4
Exit:
```

'Less Than' Test

- Besides testing for equality or inequality, it is often useful to see if a variable is less than another variable.
  - e.g., exit from a loop when the array index is less than a variable

- **`slt`** ('set on less than'):
  - **`slt  reg1, reg2, reg3`**
  - register **`reg1`** is set to 1 if the value in **`reg2`** is less than the value in **`reg3`**; otherwise, register **`reg1`** is set to 0

- **`slti`** ('set on less than immediate')
  - **`slti  $t0, $s2, 10`** # $t0=1 if $s2 < 10

Realizing Comparison Operations

- MIPS compilers use **`beq`**, **`bne`**, **`slt`**, **`slti`** and the fixed value of 0 (always available by reading register **`$zero`**) to create all comparison operations:
  - **equal**
  - **not equal**
  - **less than**
  - **less than or equal**
  - **greater than**
  - **greater than or equal**

Example

- Give the MIPS code that tests if variable **a** (corresponding to register **$s0**) is less than variable **b** (register **$s1**) and then branch to label **L** if the condition holds.

- **Answer**:

```
slt $t0, $s0, $s1   # $t0 gets 1 if $s0 < $s1
bne $t0, $zero, L   # go to L if $t0 != 0
```

- **Remark**:
  - Instead of providing a separate 'branch if less than' instruction which will complicate the instruction set, the MIPS architecture chooses to do this operation using two faster MIPS instructions – similar for other conditional branches.

Other Branch Instructions

Branch on greater than or equal to zero

- bgez $s, label # if ($s >= 0)

Branch on greater than zero

- bgtz $s, label # if ($s > 0)

Branch on less than or equal to zero

- blez $s, label # if ($s <= 0)

Branch on less than zero

- bltz $s, label # if ($s < 0)

'Jump Register' Instruction

- Another unconditional jump instruction:
  - `jr` ('jump register'):
    - e.g. `jr reg`
    - jump to address specified in register `reg`

- It is usually used for procedure call and case/switch statement

Exercise

- Consider the program below: What are the values stored in Array1 after the program is executed? (It depends on where '**jr**' goes)

```
.data
Array1: .word 4 8 12 16 20

.text
.globl __start
__start:

la $t0, Array1
lw $t1, 4($t0)
lw $t2, 8($t0)
la $s0, Label1
add $s0, $s0, $t1
jr $s0

Label1:
add $t1, $t1, $t1
add $t1, $t2, $t2
sw $t1, 12($t0)
```

i) What are the values of t1 & t2 ?

ii) If the instruction add $t1, $t1, $t1 stores at address 10000, what is the value of s0 & what jr $s0 does ?

Store at address 10000

iii) Where is this instruction stored ?

Exercise – Solution

Array1: .word 4 8 12 16 20

```
la $t0, Array1
lw $t1, 4($t0)
lw $t2, 8($t0)
```

Array1 →

Address	Value	Array element
t0	4	Array1[0]
t0+4	8	Array1[1]
t0+8	12	Array1[2]
t0+12	16	Array1[3]
t0+16	20	Array1[4]

i) So, t1 = 8, t2 = 12

```
la $s0, Label1
add $s0, $s0, $t1
jr $s0
```

ii) s0 = 10000 after `la $s0, Label1` is executed.
Hence, the next instruction to be run after `jr $s0` is stored at 10000 + 8 = 10008th byte of the memory

```
Label1:
add $t1, $t1, $t1
add $t1, $t2, $t2
sw $t1, 12($t0)
```

Label1 →

Address	Instruction
10000	add $t1, $t1, $t1
10004	add $t1, $t1, $t2
10008	sw $t1, 12($t0)

iii) All MIPS instructions are fixed as 4 bytes long. So, `sw $t1, 12($t0)` should be executed after `jr $s0` (2 instructions skipped). Array1[3] = t1 = 8 at the end

4. Dealing with "Procedure"

Supporting Procedures in Computer Hardware

- **Procedures** (also called **subroutines**) are necessary in any programming language
- They allow better structuring of programs
- Thus we need mechanisms that allow to **jump** to the procedure and to **return** from it

```
k = 0;
switch (k){
  case 0 : f = max(i,j);
           i = i + j;
           break;
  case 1:  f = max(g,h);
           i = i + j;
           break;

}
```

```
int    max(int k, int l)
       if (k <= l)
              return l;
       else
              return k;
}
```

Supporting Procedures in Computer Hardware

- Necessary steps for executing a procedure:
  1. Place the parameters in place where the procedure can get them
  2. Transfer control to the procedure
  3. Acquire the storage resources needed for the procedure
  4. Perform the desired task
  5. Place the result value in a place where the caller can access it
  6. Return control to the point of origin, since a procedure can be called from several points in a program

Registers for Procedures

- Registers for procedure calling:
  - `$a0-$a3`: four **argument registers** for passing parameters
  - `$v0-$v1`: two **value registers** for returning values
  - `$ra`: one **return address register** for returning to the point of origin

- **Program counter** (PC) or **instruction address register**:
  - Register that holds address of the current instruction being executed
  - It is updated after executing the current instruction
    - How?
    - PC = PC + 4 *or* PC = branch target address

Instructions for Procedures

- `jal` ('jump and link'):
  - `jal ProcedureAddress`
  - Two things happen at the same time
    1. First, it save the address of the following instruction (i.e., PC + 4 as return address) to register `$ra`
    2. Then, jump to address specified by `ProcedureAddress`

- `jr` ('jump register'):
  - `jr register`
  - An unconditional jump to the address specified in a register
  - Can be used to return from a procedure
    - How?

      `jr $ra` (jumps to the address stored in register `$ra`)

Caller and Callee Coordination

- **The calling program (caller)**
  - Passing parameters:
    - Puts the parameter values in `$a0` - `$a3`
    - Invokes `jal X` to jump to procedure X

- **Procedure X (callee)**
  - Performs the calculations
  - To return the results, place the results in `$v0` - `$v1`
  - Returns control to the caller using `jr $ra`
  - Caller picks up the result from `$v0` - `$v1`

Example

```
12    instruction1
16    instruction2
20    jal max
24    instruction3
```

What gets done here is

```
$ra = PC + 4 = 20 + 4 = 24
PC  = addr(max) = 60
```

```
max:  60    instruction5
      64    instruction6
      68    instruction7
      72    instruction8
      76    jr $31
```

It means "jr $ra"

Problem with Nested Procedures


```
        12      instruction1
        16      instruction2
        20      jal max
        24      instruction3
```

What gets done here is

```
$ra = PC + 4 = 20 + 4 = 24
PC  = addr(max) = 60
```

```
max:    60      instruction5
        64      instruction6
        68      jal proc
        72      instruction8
        76      jr $31
```

What if we replace instruction 7 by another procedure call, say jal proc?

```
$ra = 72
PC  = addr(proc) = 80
```

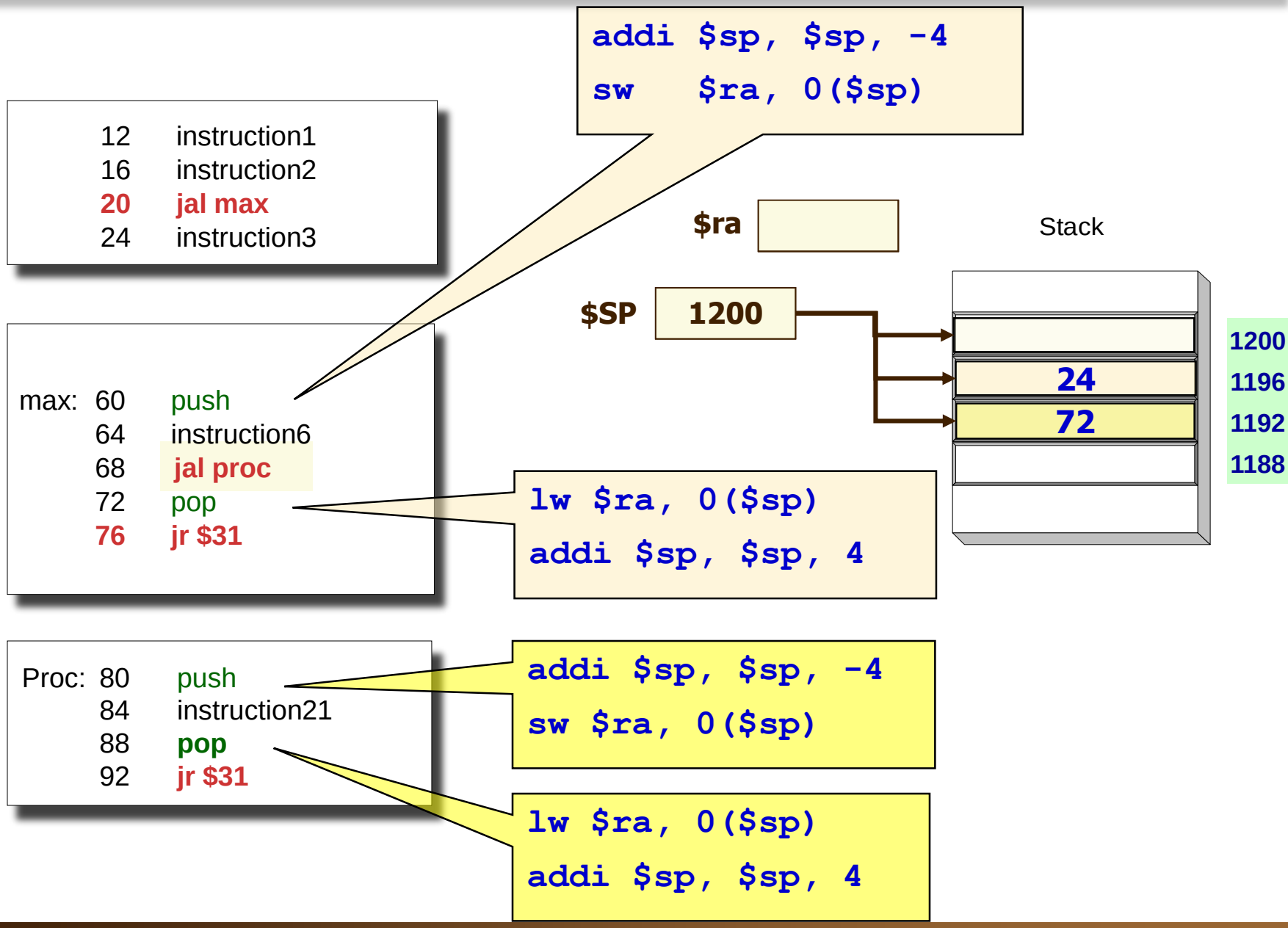
```
Proc:   80      instruction20
        84      instruction21
        88      jr $31
```

Oops! $ra = 72!!!!
Can't return to line 24

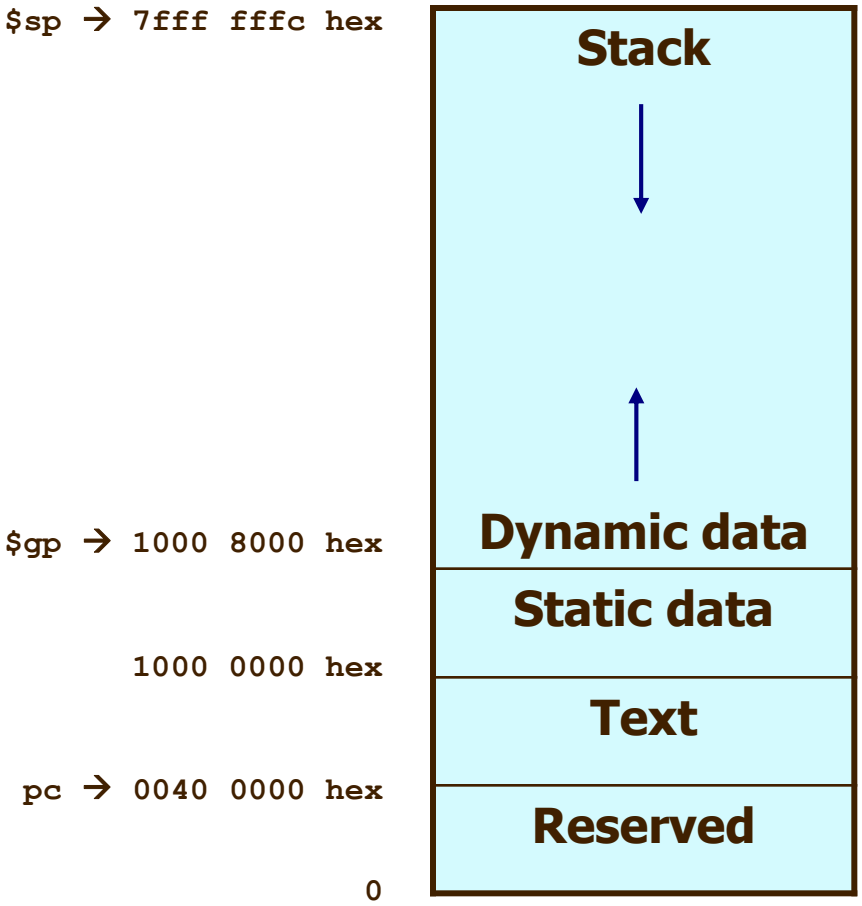
Stack: Supporting Procedures in MIPS

- Since **procedures** are like small programs themselves, they may **need to use the registers**, and they **may also call other procedures** (nested calls)
  - If we don't save some of the values stored in the registers, they will be wiped each time we call a new procedure
    - e.g. $ra was wiped out in previous example in max(), and we have no way to return from nested procedure calls
- In MIPS, we need to save the registers by ourselves (some other ISAs would do it on your behalf)
  - The perfect place for this is called a **stack**
    - a memory accessible only from the top (Last In First Out, LIFO)
    - placing things on the stack is called **push**
    - removing them is called **pop**
  - **push** and **pop** are simply **storing** and **loading** words to and from a specific location in the memory pointed to by **the stack pointer $sp** which always points to top of the stack

Using Stack to Deal with Nested Procedure



MIPS Memory Allocation



stack

For data structures that grow and shrink (e.g., linked lists) **heap**

For constant and other static variables **static data segment**

Home of MIPS machine code, or **text segment**

Heap operation:

- **malloc()** allocate space on the heap and returns a pointer to it
- **free()** releases space on the stack to which the pointer points

Summary of MIPS Architecture Revealed So Far

- **MIPS operands**:
  - 32 registers (32 bits each)
  - 2^{30} memory word locations (32 bits each)

- **MIPS instructions**:
  - Arithmetic: `add`, `sub`, `addi`
  - Data transfer: `lw`, `sw`
  - Logical: `and`, `or`, `nor`, `andi`, `ori`, `sll`, `srl`
  - Conditional branch: `beq`, `bne`, `slt`
  - Unconditional jump: `j`, `jr`, `jal`

- **MIPS instruction formats**:
  - R-format, I-format, J-format (used by `j` and `jal`; to be explained later)

MIPS Register Conventions

Name	Register number	Usage	Preserved on call?
`$zero`	0	constant value 0	n.a.
`$at`	1	reserved for assembler	n.a.
`$v0-$v1`	2-3	values for results and expression evaluation	no
`$a0-$a3`	4-7	arguments	no
`$t0-$t7`	8-15	temporaries	no
`$s0-$s7`	16-23	saved temporaries	yes
`$t8-$t9`	24-25	more temporaries	no
`$k0-$k1`	26-27	reserved for operating system kernel	n.a.
`$gp`	28	pointer to global area	yes
`$sp`	29	stack pointer	yes
`$fp`	30	frame pointer	yes
`$ra`	31	return address	yes

Preserved on call means, the value of those registers should remain the same before and after the procedure is called

If any of those registers are modified inside the procedure, you should put them into stack before the procedure is actually executed

Communication with People

- Most computers use 8-bit (**bytes**) to represent characters
  - ASCII: American Standard Code for Information Interchange
  - Example:

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
48	0	49	1	65	A	66	B	90	Z
97	a	98	b	32	**Space**	35	#	42	*

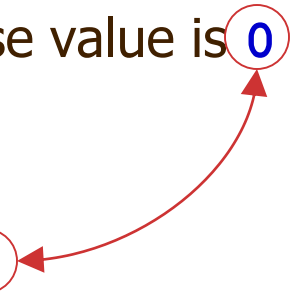
- Notice that character "a" and "A" are assigned with different values!

- Operation with byte: **`lb`** (load byte), **`sb`** (store byte)
  - Example

```
lb $t0, 0($sp)      # Read byte from source
sb $s0, 0($gp)      # Write byte to destination
```

- **Characters are normally combined into strings**

- How to represent a string? Three choices are:
  1. First position of a string is reserved to give the length of a string
  2. An accompanying variable has the length of the string (as in a structure)
  3. The last position of a string is indicated by a character used to mark the end of a string
     - "C" uses the 3rd choice
     - "C" terminates a string with a byte whose value is `0` (`null` in ASCII)
     - Example
       the string '`Cal`' → ASCII `67,97,108,0`



Example: String Copy

- Procedure **`strcpy()`** in "C" language
  - copies string **`y`** to string **`x`** using the null byte termination convention

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i])!='\0')    /* copy & test byte */
        i += 1;
}
```

MIPS Assembly Code


```
strcpy:
   addi  $sp, $sp, -4          # adjust stack for 1 more item
   sw    $s0, 0($sp)           # save $s0
   add   $s0, $zero, $zero     # i = 0 + 0
L1:
   add   $t1, $s0, $a1         # address of y[i] in $t1
   lb    $t2, 0($t1)           # $t2 = y[i]
   add   $t3, $s0, $a0         # address of x[i] in $t3
   sb    $t2, 0($t3)           # x[i] = $t2
   beq   $t2, $zero, L2        # if y[i]==0, go to L2
   addi  $s0, $s0, 1           # i = i + 1
   j     L1                    # go to L1
L2:                            # y[i] == 0: end of string;
   lw    $s0, 0($sp)           # restore old $s0
   addi  $sp, $sp, 4           # pop 1 word off the stack
   jr    $ra                   # return
```

don't have to multiply i by 4 since x and y are arrays of bytes, not of words

Dealing with 32-bit Immediate

- Constants are frequently short and fit into 16-bit field
- But sometimes they are bigger than 16 bits, e.g. 32-bit constant

Problem:

- With instruction learned so far, we cannot set registers' upper 16bits!

Solution:

- `lui` ("load upper immediate")
  - e.g. `lui reg, constant`
  - set the upper 16 bits of register `reg` to the 16-bit value specified in `constant`
  - **Set the lower 16 bits of register `reg` to zeros**
  - note that `constant` should not greater than 2^{16}

Example: Loading a 32-bit Constant

- How to load the 32-bit constant below into register $s0?

  0000 0000 0011 1101 0000 1001 0000 0000_2 (0x003D0900)

- Solution: (assuming the initial value in `$s0` is 0)

```
lui $s0, 61              # 61_10 = 0000 0000 0011 1101_2
# value of $s0 becomes 0000 0000 0011 1101 0000 0000 0000 0000_2
ori $s0, $s0, 2304       # 2304_10 = 0000 1001 0000 0000_2
# now, we get the value desired into the register
```

5. Addressing Modes

MIPS Addressing Modes

- We have seen, so far three addressing modes of MIPS:
  1. Immediate addressing: provides fast access of small **constants**

     e.g. `addi $t0, $t0, 1023`
  2. Register addressing: the operand is available in a register

     e.g. `add $t0, $t0, $t1`
  3. Base addressing: the operand is the sum of a (**base**) register and a **displacement**

     e.g. `lw $t0, 1024($t1)`

- MIPS architecture provides two more ways of addressing

J-type Instruction Format

J-type or **J-format**

- Also called pseudodirect addressing; the simplest addressing mode
- Used by instructions such as `j` ('jump') and `jal` ('jump and link')
  - e.g. `j  L1  # go to instruction labeled L1`
- 26-bit word address, corresponding to a 28-bit byte address, is concatenated with the 4 upper bits of the PC to form a 32-bit byte branch target address that corresponds to label `L1`

Example:

- If PC = 0100 1101 0111 0000 1100 0011 0000 1100
- Target L1 is: 0100 **1100 0011 1000 0001 1111 0011 0100**

Stretching the Maximum Possible Distance

Because,

- All MIPS instructions are 4 bytes long

So,

- A branch target or offset can refer to number of words instead of number of bytes

⇒ essentially stretch the maximum possible branching distance by 4x

Questions:

- What is the range a 'j' and 'jal' can jump to?
  - Within 256MB
- What if we want to jump beyond 256MB?

Stretching the Maximum Possible Distance

0x0 0000000	
…	
0x0 FFFFFFF	
0x1 0000000	
…	
0x1 FFFFFFF	
0x2 0000000	J L1
…	
0x2 FFFFFFF	L1: …
…	
0x7 0000000	
…	
0x7 FFFFFFF	

Stretching the Maximum Possible Distance

0x0 0000000	
…	
0x0 FFFFFFF	
0x1 0000000	J L1
…	
0x1 FFFFFFF	
0x2 0000000	L1: …
…	
0x2 FFFFFFF	
…	
0x7 0000000	
…	
0x7 FFFFFFF	

What if the Jump target is more than 256 MB away?

Addressing in Conditional Branches

We know that

- Conditional branch instructions (e.g., **`beq`**, **`bne`**) use I-format
- I-format can only specify 16-bit addresses

How to branch?

- PC-relative addressing
  - A branch offset is added to (PC+4) to obtain address to branch to
    - Branch offset is described in number of words. Why?
  - Branching within 2^{15} words before or after the current instruction is possible
  - This is good enough since conditional branches tend to branch to a nearby instruction

Notes:

While an instruction is being executed, the PC always points to the current instruction, i.e., address of current instruction

Example: Branch Offset in Machine Language

- Traditional loop in "C":

```
while (save[i] == k)
    i +=1 ;
```

- Assume `$s3` → `i`, `$s6` → `save`, `$s5` → `k`, loop start at memory location 80000
- MIPS assembly code

```
Loop:
   sll  $t1, $s3, 2
   add  $t1, $t1, $s6
   lw   $t0, 0($t1)
   bne  $t0, $s5, Exit
   addi $s3, $s3, 1
   j    Loop
Exit:
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	Exit:....					

Branching Far Away

- From 16-bit word address to 26-bit word address:
  - replace this

```
beq $s0, $s1, L1   # L1 = 16-bit address
```

  - with this

```
bne $s0, $s1, L2   # L2 = 16-bit address
j   L1             # L1 = 26-bit address
L2:
```

Attention:

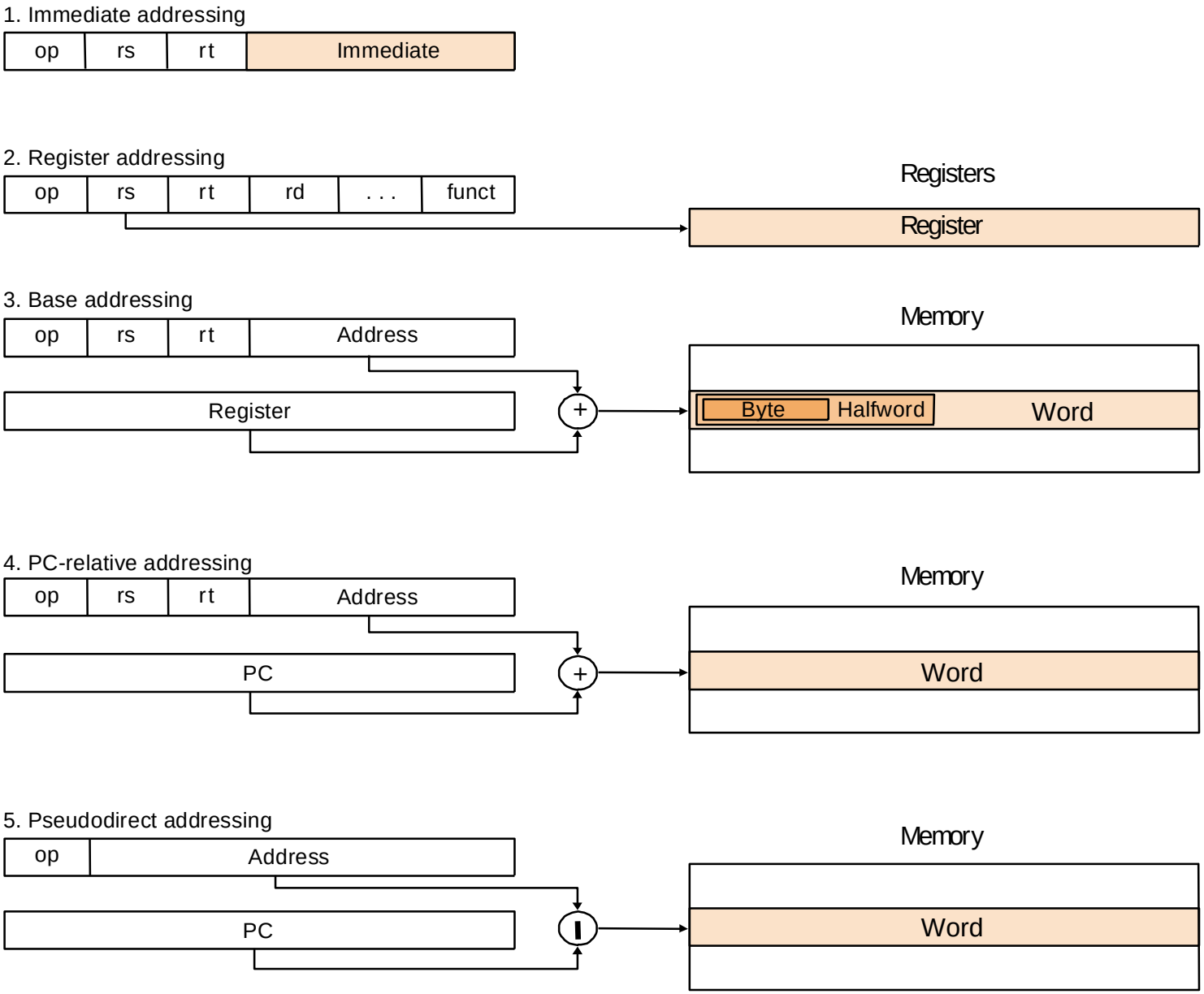
The tradeoff is longer program execution time

- i.e. need to execute two instructions rather than just one

Summary of MIPS Addressing Modes

1. **Immediate addressing**
   - The operand is a constant within the instruction itself
2. **Register addressing**
   - The operand is a register
3. **Base addressing** or **displacement addressing**
   - The operand is at the memory location with address = (register) + constant
4. **PC-relative addressing**
   - The address is = (PC) + 4 + constant
5. **Pseudodirect addressing**
   - The jump address is a constant in the instruction concatenated with the upper 4 bits of the PC

Summary of MIPS Addressing Modes (cont'd)



Summary of MIPS Architecture Revealed So Far

- **MIPS operands**:
  - 32 registers (32 bits each)
  - 2^{30} memory word locations (32 bits each)

- **MIPS instructions**:
  - Arithmetic: **`add`**, **`sub`**, **`addi`**
  - Data transfer: **`lw`**, **`sw`**, **`lb`**, **`sb`**, **`lui`**
    - **`lb`** and **`sb`** are similar to **`lw`** and **`sw`**, but for transferring bytes instead of words
  - Conditional branch: **`beq`**, **`bne`**, **`slt`**, **`slti`**
  - Unconditional jump: **`j`**, **`jr`**, **`jal`**

- **MIPS instruction formats**:
  - R-format, I-format, J-format

6. Other Issues

Alternatives to the MIPS Approach

- MIPS is an example of **RISC**
  - **Reduced Instruction Set Computer**
  - Each instruction does one simple thing
  - Most existing processors are RISC since it is more promising

- Another approach is **CISC**
  - **Complex Instruction Set Computer**
  - One instruction may do multiple things, e.g. Intel's instruction set

	RISC	**CISC**
Number of instructions in a program	(–) more	(+) less
Time to execute the program	(+) usually less	(–) usually more
Hardware design	(+) simple	(–) complex

(+) means advantage, (–) means disadvantage

Comparing RISC and CISC in Details

RISC	CISC
Through quantitative measurements, choose only the most useful instructions and addressing modes.	Choose instructions and addressing modes that make the translation of high-level languages to assembly language simpler.
With few instructions and addressing modes, we can directly execute them in hardware.	Since we can have many instructions and addressing modes, we need a **microcode** (or **microprogrammed control**) to execute them in hardware.
A lot of chip space can be left for a large number of registers and cache memory.	We can have only few registers and small cache memory.
Compilers are more difficult to write.	Compilers are easier to write.
Assembly language programs are more difficult to write.	Assembly language programs are easier to write.

Pseudoinstructions

- Assembly language instructions that do not have corresponding machine instructions (i.e., they need not be implemented directly in hardware)

Why Pseudoinstructions?

- Their appearance in assembly language simplifies programming and translation, giving MIPS a richer set of assembly language instructions than those implemented by the hardware.

Cost of supporting Pesudoinstructions

- The only cost is reserving one register, `$at`, for use by the assembler

Examples of Pseudoinstructions

- **`move`**:
  - `move $t0, $t1  # $t0 gets value of $t1`
  - The assembler converts this pseudoinstruction into the machine language equivalent of the following instruction:

    `add  $t0, $zero, $t1  # $t0 gets 0 + value of $t1`

- Others:
  - **`blt`** ('branch on less than')
  - **`ble`** ('branch on less than or equal')
  - **`bgt`** ('branch on greater than')
  - **`bge`** ('branch on greater than or equal')

Key Concepts to Remember

- The stored-program concept underlies today's digital computers
- An instruction specifies an operation and its corresponding operand(s)
- All MIPS instructions are 32 bits in length
  - To simplify the instruction set architecture
  - But, multiple instruction formats are supported
- Registers are fast temporary storage inside the processor
- Four design principles for ISA
  - Simplicity favors regularity
  - Smaller is faster
  - Make common case fast
  - Good design demands good compromises

Key Concepts to Remember (cont'd)

- Program counter is a special register
  - Pointing to the current instruction to be fetched and executed
- Branch/jump instructions often require branch address calculation
- MIPS supports different addressing modes
  1. Register
  2. Displacement
  3. Immediate
  4. PC-relative
  5. Pseudodirect
- Pseudoinstructions extend the MIPS instruction set
  - To facilitate program development
- RISC and CISC are two very different design philosophies