CO20-320241

Computer Architecture and Programming Languages

CAPL

Lecture 12 & 13

Dr. Kinga Lipskoch

Fall 2019

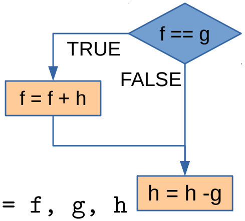
Branch & Jump

- Two alternatives
  - Continue if “condition” is false
  - Go to another program segment
- Conditional branching: branch after register comparison:
  - Equal `beq register1, register2, LABEL_NAME`
  - Not equal `bne register1, register2, LABEL_NAME`
    - `register1` and `register2` are compared
    - `LABEL_NAME` is the target address that is used as next instruction address if the condition is true
- Unconditional branching: `jump`
  - Jump version 1: `j address`
  - Jump version 2: `jr $s3      # address in reg. #3`

Example: `if`

- C-code example:

```
if (f == g)
    f = f + h;
h = h - g;
```



- Register assignment: `$s0, $s1, $s2 = f, g, h`

```
     bne $s0, $s1, AFTER_IF       # branch
     add $s0, $s0, $s2            # skipped
AFTER_IF: sub $s2, $s2, $s1      # always ex.
```

- Reminder:
  - Registers for variables `$s0` … `$s7` 16 … 23
  - Registers for temp. variables `$t0` … `$t7` 8 … 15
  - Register `$zero` always 0

Example: if then else

- C-code example:

```c
if (i != j)
  h = i + j;
else
  h = i - j;
```

```
      beq $s3, $s4, ELSE
      add $s2, $s3, $s4
      j AFTER_IF_ELSE
ELSE: sub $s2, $s3, $s4
AFTER_IF_ELSE: ...
```

- Register assignment:
  `$s0, $s1, $s2, $s3, $s4 = f, g, h, i, j`

while Loop

- A while loop in C:

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

- Assumption: $s3= i, $s5 = k, base of array saved in $s6

```
LOOP: add  $t1, $s3, $s3      # $t1 = 2 * i
      add  $t1, $t1, $t1      # $t1 = 4 * i
      add  $t1, $t1, $s6      # $t1 = addr.
      lw   $t0, 0($t1)        # load
      bne  $t0, $s5, EXIT     # exit if !=
      addi $s3, $s3, 1        # i = i + 1
      j    LOOP
EXIT: ...
```

Other Decisions: set less than

- There is no explicit branch, rather two fast instructions
- Set `register1` on `register2` less than `register3`
  - `slt register1, register2, register3`
- Compares two registers, `register2` and `register3`
- If the second is less than the third
  - `register1 = 1` else
  - `register1 = 0`
- `slt $t0, $s1, $s2`
- Equivalent to this C-code

```c
if ($s1 < $s2)
  $t0 = 1
else
  $t0 = 0
```

Example: `if (f < g) ...`

-
  ```
  if (f < g)
      h = h + 1;
  else
      h = h - 1;
  ```
- Assumption: `f`, `g`, `h` stored in `$s1`, `$s2`, `$s3`

```
        slt $t0, $s1, $s2
        beq $t0, $0, ELSE
        addi $s3, $s3, 1
        j AFTER_IF_ELSE
ELSE: addi $s3, $s3, -1
AFTER_IF_ELSE: ...
```

Example: `switch` (1)

-
  ```
  switch(k) {
    case 0:     f = i + j; break;
    case 1:     f = g + h; break;
    case 2:     f = g - h; break;
    case 3:     f = i - j; break;
  }
  ```
  Assumption:
  `f, g, h, i, j, k = $s0, $s1, $s2, $s3, $s4, $s5`
- Jump table, beginning at `$t4`

```
$t4          address of CASE0
$t4 + 8      address of CASE1
$t4 + 16     address of CASE2
$t4 + 24     address of CASE3
```

Example: `switch` (2)

- Check range of `k` (`k >= 0` and `k < 4`)

```
slt    $t3, $s5, $zero
bne    $t3, $zero, exit
slti   $t3, $s5, 4
beq    $t3, $zero, exit
```

`$zero = $0 = Register 0 = constant 0`

- Index address to byte address

```
sll $t1, $s5, 3      # $t1 = 8 * k
```

- Jump table, beginning at `$t4`, length $= 2$ words

```
add $t1, $t1, $t4
jr  $t1
```

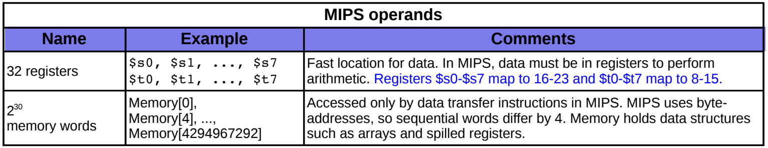
jump to instructions from the jump table

Example: `switch` (3)

Switch as jump table

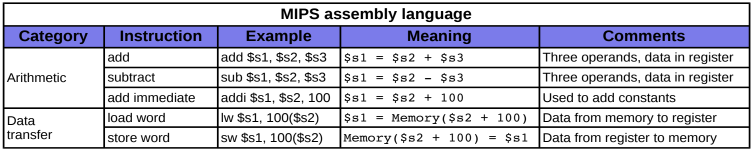
```
CASE0:    add $s0, $s3, $s4
        j    exit
CASE1:    add $s0, $s1, $s2
        j    exit
CASE2:    sub $s0, $s1, $s2
        j    exit
CASE3:    sub $s0, $s3, $s4
exit: ...                    # next statement
```

First Overview



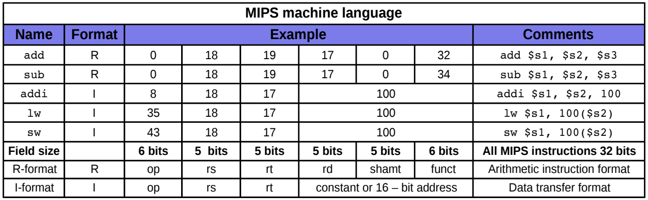
MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, ..., $s7` `$t0, $t1, ..., $t7`	Fast location for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16-23 and $t0-$t7 map to 8-15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte-addresses, so sequential words differ by 4. Memory holds data structures such as arrays and spilled registers.



MIPS assembly language

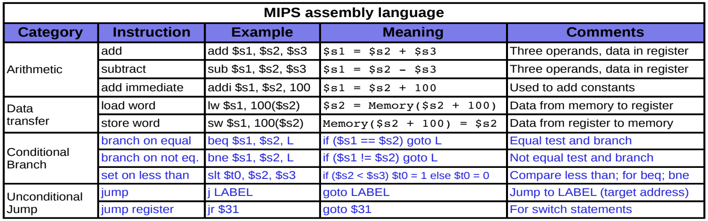
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	`$s1 = $s2 + $s3`	Three operands, data in register
	subtract	sub $s1, $s2, $s3	`$s1 = $s2 - $s3`	Three operands, data in register
	add immediate	addi $s1, $s2, 100	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	`$s1 = Memory($s2 + 100)`	Data from memory to register
	store word	sw $s1, 100($s2)	`Memory($s2 + 100) = $s1`	Data from register to memory



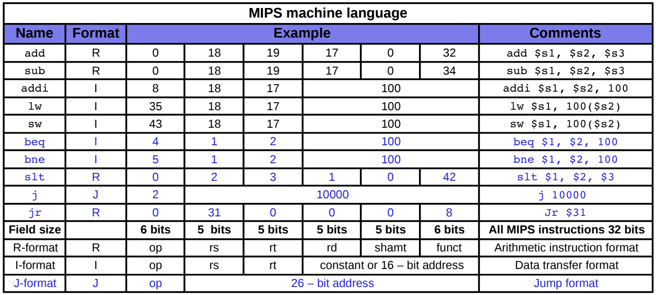
MIPS machine language

Name	Format	Example						Comments
`add`	R	0	18	19	17	0	32	`add $s1, $s2, $s3`
`sub`	R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
`addi`	I	8	18	17	100			`addi $s1, $s2, 100`
`lw`	I	35	18	17	100			`lw $s1, 100($s2)`
`sw`	I	43	18	17	100			`sw $s1, 100($s2)`
Field size		**6 bits**	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**	**All MIPS instructions 32 bits**
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	constant or 16 – bit address			Data transfer format

Second Overview



MIPS assembly language				
Category	**Instruction**	**Example**	**Meaning**	**Comments**
Arithmetic	add	add $s1, $s2, $s3	`$s1 = $s2 + $s3`	Three operands, data in register
	subtract	sub $s1, $s2, $s3	`$s1 = $s2 - $s3`	Three operands, data in register
	add immediate	addi $s1, $s2, 100	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	`$s2 = Memory($s2 + 100)`	Data from memory to register
	store word	sw $s1, 100($s2)	`Memory($s2 + 100) = $s2`	Data from register to memory
Conditional Branch	branch on equal	beq $s1, $s2, L	if ($s1 == $s2) goto L	Equal test and branch
	branch on not eq.	bne $s1, $s2, L	if ($s1 != $s2) goto L	Not equal test and branch
	set on less than	slt $t0, $s2, $s3	if ($s2 < $s3) $t0 = 1 else $t0 = 0	Compare less than; for beq; bne
Unconditional Jump	jump	j LABEL	goto LABEL	Jump to LABEL (target address)
	jump register	jr $31	goto $31	For switch statements



MIPS machine language								
Name	**Format**	**Example**						**Comments**
`add`	R	0	18	19	17	0	32	`add $s1, $s2, $s3`
`sub`	R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
`addi`	I	8	18	17	100			`addi $s1, $s2, 100`
`lw`	I	35	18	17	100			`lw $s1, 100($s2)`
`sw`	I	43	18	17	100			`sw $s1, 100($s2)`
`beq`	I	4	1	2	100			`beq $1, $2, 100`
`bne`	I	5	1	2	100			`bne $1, $2, 100`
`slt`	R	0	2	3	1	0	42	`slt $1, $2, $3`
`j`	J	2	10000					`j 10000`
`jr`	R	0	31	0	0	0	8	`Jr $31`
Field size		**6 bits**	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**	**All MIPS instructions 32 bits**
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	constant or 16 – bit address			Data transfer format
J-format	J	op	26 – bit address					Jump format

Procedure Calls

- `int my_proc(int g, int h, int i, int j)`
- Procedures & subroutines for frequently used code segments
- Structuring of code, code reuse
- Steps
  1. Define a joint parameter space for main program and the procedure
  2. Transfer control to procedure
  3. Acquire the required storage resources
  4. Perform the desired task
  5. Store parameters in the joint space
  6. Return control to the caller

Register Assignment Convention

- `$a0 ... $a3` four argument registers for passing parameters
- `$v0 ... $v1` two return value registers
- `$ra` return address register register #31
- Use of arguments and value registers:
  - task of the compiler
- Handling of control passing mechanism:
  - task for the machine
- Instruction: `jal` - jump and link

Jump and Link

- Begin of procedure:

  `jal ProcedureAddress`
- Operation:
  - Jump to the address handed as parameter
  - Return address is placed in link register
  - Link register is `$31` (`$ra`)
- Return:

  `jr $ra`
- Return register holds the value PC + 4
  - PC is the Program Counter, contains the address of the instruction that is currently executed
- Storing of additional variables: stack

Calling a Procedure

- Caller puts parameter values into `$a0 - $a3`
- `jal X` (to jump to procedure `X`, the callee)
- Computations inside callee
- Callee places results into `$v0 - $v1` and returns control to caller using `jr $ra`

Spilling Registers

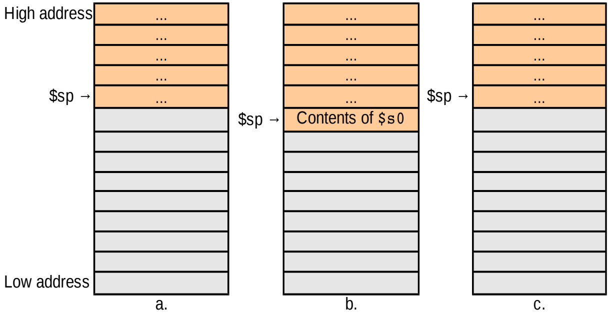
- What happens if procedure wants to pass more than 4 registers?
  - Registers needed by caller must be restored to the value before the procedure was invoked
  - Spill registers (save less used and later needed registers to memory)
- Implemented by stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)
  - `$sp` is used to address the stack
  - Points to most recently allocated address or where next procedure should place the content of registers
  - Points to the top of stack

Stack

- The stack is located in memory
- Placing data onto the stack: push
- Removing data from stack: pop
- Stack pointer `$sp` is adjusted for each register that is saved or restored
- Grows from high address to low address
- push (save) $\rightarrow$ subtract from stack pointer
- pop (restore) $\rightarrow$ add to stack pointer

Pushing and Popping the Stack

a. `$sp` points to top of stack
b. One register has been pushed onto stack, `$sp` has been decremented
c. Register has been popped from stack, `$sp` has been incremented accordingly



Example: Simple Function

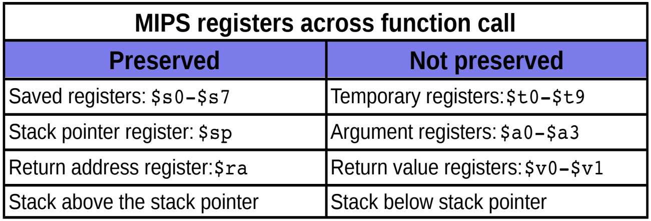
```
int leaf_example(int g,int h,int i,int j) {
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

```
# example assumes: $s0 has to be used in the function
leaf_example:
    addi $sp, $sp, -4   # adjust stack,
                        # make room for 1 var
    sw   $s0, 0($sp) # save register $s0 for later use
    add  $t0, $a0, $a1   # reg $t0 contains g + h
    add  $t1, $a2, $a3   # reg $t1 contains i + j
    sub  $s0, $t0, $t1   # f = $t0 - $t1
    add  $v0, $s0, $zero   # put return val into reg
    lw   $s0, 0($sp)   # restore reg $s0 for caller
    addi $sp, $sp, 4   # adjust stack to delete 1 item
    jr   $ra   # jump back to calling routine
```

Who Saves the Register?

- Caller save
  - All values that have to be kept must be saved before a procedure is called
  - Convention: callee does not preserve temporary registers, thus caller has to save `$t0 - $t9` before calling procedure in case they are to be reused
- Callee save
  - Within the procedure all used registers are saved and afterwards restored
  - Callee will preserve `$s0 - $s7` (if used callee saves and restores these registers)
- Convention valid for the whole OS
  - Calls written in different languages can be mixed C, Fortran, Assembler

Which Registers are Saved?



MIPS registers across function call	
Preserved	**Not preserved**
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below stack pointer

`$sp` is preserved because the function has to make sure that `$sp` is added and subtracted the same amount within the function

Recursive Functions (1)

C-code for the factorial function:

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

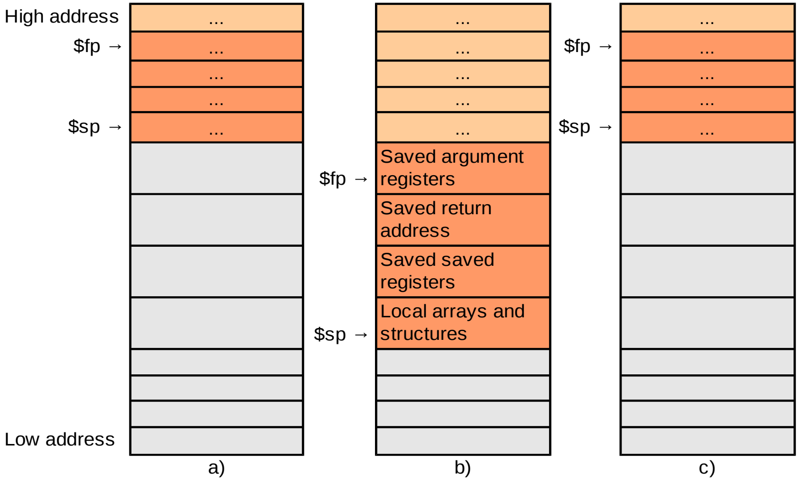
Recursive Functions (2)

```
fact:
  addi $sp, $sp, -8   # adjust stack for two items
  sw   $ra, 4($sp)    # save the return address
  sw   $a0, 0($sp)    # save the argument n
  slti $t0, $a0, 1    # test for n < 1
  beq  $t0, $zero, R_STEP # if (n >= 1) goto R_STEP
  addi $v0, $zero, 1  # set return val to 1
  addi $sp, $sp, 8    # pop 2 items off stack
  jr   $ra            # return to after jal
R_STEP:                   # Label
  addi $a0, $a0, -1   # argument gets (n - 1)
  jal  fact           # call fact with (n - 1)
  lw   $a0, 0($sp)    # ret from jal: restore arg n
  lw   $ra, 4($sp)    # restore return address
  addi $sp, $sp, 8    # pop 2 items off stack
  mul  $v0, $a0, $v0  # return n * fact(n - 1)
  jr   $ra            # return to caller
```

Frame Pointer (1)

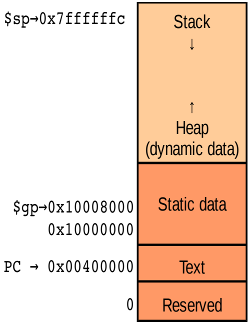
- The stack is also used to store variables that are local to function, but do not fit into registers
  - local arrays, structures
- The segment of the stack containing function's saved registers is called procedure frame or function frame
- A frame pointer (`$fp`) points to the first word of the frame of a function
- `$sp` might change during function
- `$fp` is a stable base register within a function for local memory references

Frame Pointer (2)



Memory: Heap

- Stack starts at high end and grows downwards
- First part of low end is reserved
- Then text segment
  - executable machine code
- Above static data segment
  - constants and variables
- Dynamic data such as arrays in lists are placed in the heap
  - `malloc()`, `free()`
- Thus stack and heap grow towards each other



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	`$s1 = $s2 + $s3`	Three operands, data in register
	subtract	sub $s1, $s2, $s3	`$s1 = $s2 – $s3`	Three operands, data in register
	add immediate	addi $s1, $s2, 100	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	`$s2 = Memory($s2 + 100)`	Data from memory to register
	store word	sw $s1, 100($s2)	`Memory($s2 + 100) = $s2`	Data from register to memory
Conditional Branch	branch on equal	beq $1, $2, L	if ($1 == $2) goto L	Equal test and branch
	branch on not eq.	bne $1, $2, L	if ($1 != $2) goto L	Not equal test and branch
	set on less than	slt $t0, $s2, $s3	if ($s2 < $s3) $t0 = 1 else $t0 = 0	Compare less than; for beq; bne
Unconditional Jump	jump	j LABEL	goto LABEL	Jump to LABEL (target address)
	jump register	jr $ra	goto $ra	For switch, procedure return
	jump and link	jal LABEL	$31 = PC + 4; goto LABEL	For procedure call

MIPS machine language

Name	Format	Example						Comments
`add`	R	0	18	19	17	0	32	`add $s1, $s2, $s3`
`sub`	R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
`addi`	I	8	18	17	100			`addi $s1, $s2, 100`
`lw`	I	35	18	17	100			`lw $s1, 100($s2)`
`sw`	I	43	18	17	100			`sw $s1, 100($s2)`
`beq`	I	4	1	2	100			`beq $1, $2, 100`
`bne`	I	5	1	2	100			`bne $1, $2, 100`
`slt`	R	0	2	3	1	0	42	`slt $1, $2, $3`
`j`	J	2	10000					`j 10000`
`jr`	R	0	31	0	0	0	8	`jr $31`
`jal`	J	3	10000					`jal 10000`
Field size		**6 bits**	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**	**All MIPS instructions 32 bits**
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	constant or 16 – bit address			Data transfer format
J-format	J	op	26 – bit address					Jump format

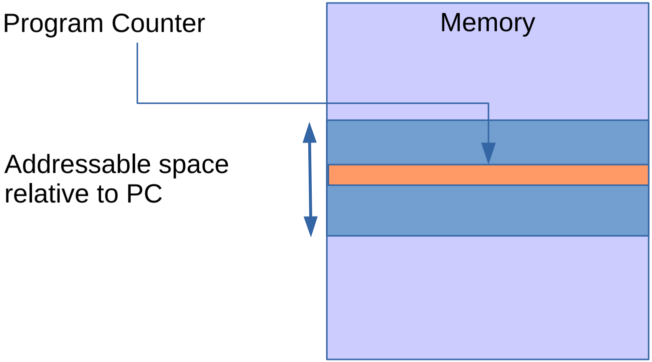
Limitations in Addressing

- Jump register
  - 32 bit register - 32 bit address
- Jump address
  - 6 bits for the jump op-code
  - 26 bits for the address
- Branch
  - 16 bits for the branch op-code & registers
  - 16 bits for the address
  - First idea: limitation of branch space to the first 2^{16} bits
- Word address instead of byte address $\rightarrow$ additional 2 bits, rest comes from PC

Relative Addressing (1)

- Combination of a base register and the address in the branch operation
- PC = PC + branch address
- Reference is the Program Counter, PC
- Relative jumps & branches
- No serious restriction
- High probability of the target being in the range of PC
  - 50% of branch destinations in SPEC benchmark are less than 16 instructions away

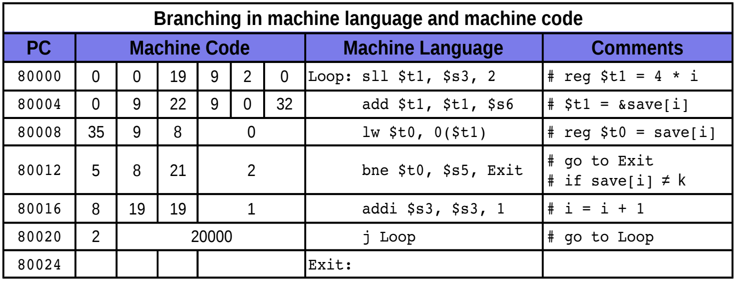
Relative Addressing (2)



For larger distances: Jump register required

Addressing in Branches and Jumps

- Branches use PC - relative addressing
  - destination = PC + 4 + word address
- Jump uses also word addressing
  - first 4 bits of PC
  - destination = PC[0 : 3] + word address (+=concatenate)



Branching in machine language and machine code

PC	Machine Code						Machine Language	Comments
80000	0	0	19	9	2	0	`Loop: sll $t1, $s3, 2`	`# reg $t1 = 4 * i`
80004	0	9	22	9	0	32	`add $t1, $t1, $s6`	`# $t1 = &save[i]`
80008	35	9	8	0			`lw $t0, 0($t1)`	`# reg $t0 = save[i]`
80012	5	8	21	2			`bne $t0, $s5, Exit`	`# go to Exit` `# if save[i] ≠ k`
80016	8	19	19	1			`addi $s3, $s3, 1`	`# i = i + 1`
80020	2	20000					`j Loop`	`# go to Loop`
80024							`Exit:`	

Branching Far Away

- Replace

```
beq $s0, $s1, L1
```

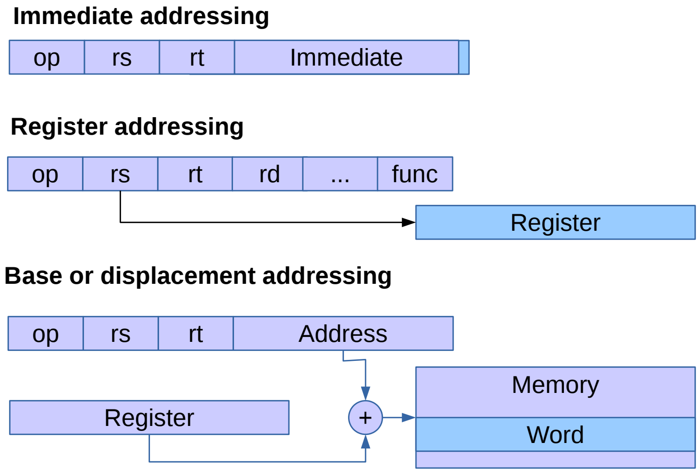
- By pair of instructions:

```
    bne  $s0, $s1, L2
    j    L1
L2: ...
    ...
```

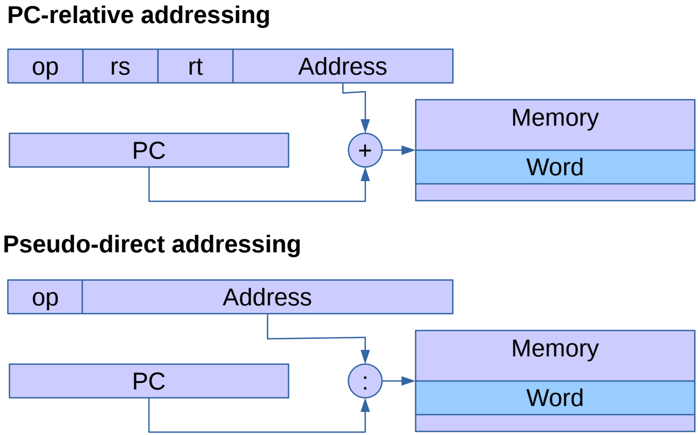
Addressing Modes (1)

- Immediate addressing, `addi`
  - Operand is constant
- Register addressing, e.g., `add`
  - Operand is a register
- Base or Displacement addressing, e.g., `lw`
  - Operand is in the memory
  - Address is sum of register + address in instruction
- PC-relative addressing, e.g., `beq` (branch)
  - address is sum of PC and constant of instruction (16-bit address shifted left 2 bits)
- Pseudo-direct addressing `j`
  - jump address is the 26 bits shifted left 2 bits of the instruction concatenated with 4 upper bits of PC

Addressing Modes (2)



Addressing Modes (3)



Non-MIPS Addressing Modes

- MIPS:
  - add register register register
  - add register register immediate
- Other:
  - add register, register, memory
  - add register, memory, memory
  - add memory, memory, memory
  - ...
- Memory access:
  - immediate
  - relative
  - indexed

Remarks

- Various instruction types (selection)
  - Logical operations
  - Arithmetic operations
  - Branches & jumps
- Addressing modes
  - Immediate
  - Register
  - Relative (register or PC)