MIPS Instructuction Set Architecture 2

2020-21/1

CSCM601252 - Pengantar Organisasi Komputer

Instructor: Erdefi Rakun dan Tim Dosen POK

Fasilkom UI

Outline

- MIPS Instructions Classification
- R-Format
- I-Format
- Instruction Address
- Branches: PC-Relative Addressing
- J-Format
- Addressing for Branch
- Addressing for Jump
- Addressing Modes

Note: These slides are taken from Aaron Tan's slide

MIPS Instructions Classification

- Instructions are classified according to their operands; instructions with same operand types have same representation
- R-format (Register format)
  - **add**, **sub**, **and**, **or**, **nor**, **slt** require 2 source registers and 1 destination register
  - **srl**, **sll** also belong here
- I-format (Immediate format)
  - **addi**, **subi**, **andi**, **ori**, **slti**, **lw**, **sw**, **beq**, **bne** require 1 source register, 1 immediate and 1 destination register
- J-format (Jump format)
  - **j** instruction requires only one immediate

Registers

- We shall use register numbers ($0, $1, …, $31) instead of names to simplify association.

Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

R-Format (1/2)

- Define fields of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

- A field is viewed as 5- or 6-bit unsigned integer, not as part of 32-bit integer
  - 5-bit fields can represent any number 0-31
  - 6-bit fields can represent any number 0-63

R-Format (2/2)

- **opcode**
  - partially specifies what instruction it is
  - equal to 0 for all R-Format instructions
- **funct**
  - combined with **opcode** exactly specifies the instruction
- **rs** (Source Register)
  - used to specify register containing first operand
- **rt** (Target Register)
  - used to specify register containing second operand (note that name is misleading)
- **rd** (Destination Register)
  - used to specify register which will receive result of computation
- **shamt**
  - amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31)
  - set to 0 in all but the shift instructions

R-Format: Example (1/3)

- MIPS instruction:

  add $8, $9, $10 (or **add $t0, $t1, $t2**)

  opcode = 0 (textbook pg 60 - 68)

  funct = 32 (textbook pg 60 - 68)

  rd = 8 (destination)

  rs = 9 (first operand)

  rt = 10 (second operand)

  shamt = 0 (not a shift)

R-Format: Example (2/3)

- MIPS instruction:

  add $8, $9, $10 (or **add $t0, $t1, $t2**)

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	32

Field representation in binary:

000000	01001	01010	01000	00000	100000

Hexadecimal representation of instruction:

$012A\ 4020_{16}$

R-Format: Example (3/3)

- MIPS instruction:

  sll $8, $9, 4 (or **sll $t0, $t1, 4**)

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	0	9	8	4	0

Field representation in binary:

000000	00000	01001	01000	00100	000000

Hexadecimal representation of instruction:

$0009\ 4100_{16}$

Try It Yourself #1

- MIPS instruction:

  add $10, $7, $5

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
?	?	?	?	?	?

Field representation in binary:

Hexadecimal representation of instruction:

???

I-Format (1/4)

- What about instructions with immediates?
  - 5-bit **shamt** field only represents numbers up to the value 31
  - immediates (for example for **lw**, **sw** instructions) may be much larger than this
- Compromise: Define new instruction format partially consistent with R-format:
  - If instruction has immediate, then it uses at most 2 registers

I-Format (2/4)

- Define fields of the following number of bits each: 6 + 5 + 5 + 16 = 32

6	5	5	16

- Again, each field has a name:

opcode	rs	rt	immediate

- Only one field is inconsistent with R-format.
  - **opcode**, **rs**, and **st** are still in the same locations.

I-Format (3/4)

- **opcode**
  - same as before except that, since there is no **funct** field, **opcode** uniquely specifies an instruction
- **rs**
  - specifies the source register operand (if any)
- **rt**
  - specifies register to receive result
  - note the difference from R-format instructions

I-Format (4/4)

- The **immediate** field:
  - Treated as a signed integer
  - 16 bits → can be used to represent a constant up to 2^{16} different values
  - This is large enough to handle the offset in a typical **lw** or **sw**, plus a vast majority of values that will be used in the **addi**,**subi**, **slti** instructions

I-Format: Example (1/2)

- MIPS instruction:

  addi $21, $22, -50

  opcode = 8

  rs = 22 (register containing operand)

  rt = 21 (target register)

  immediate = -50 (by default, this is decimal)

I-Format: Example (2/2)

- MIPS instruction:

  addi $21, $22, -50

Field representation in decimal:

8	22	21	-50

Field representation in binary:

001000	10110	10101	1111111111001110

Hexadecimal representation of instruction:

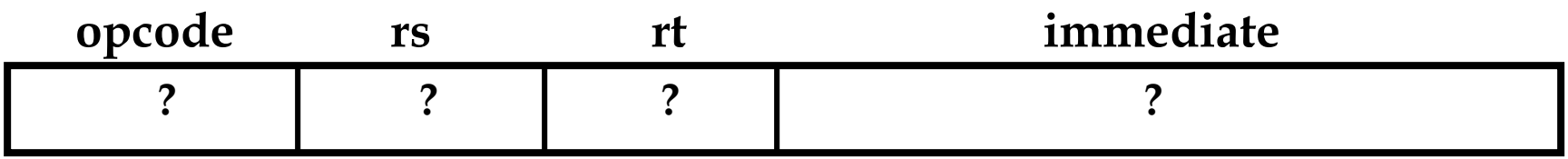
22D5 FFCE$_{16}$

Try It Yourself #2

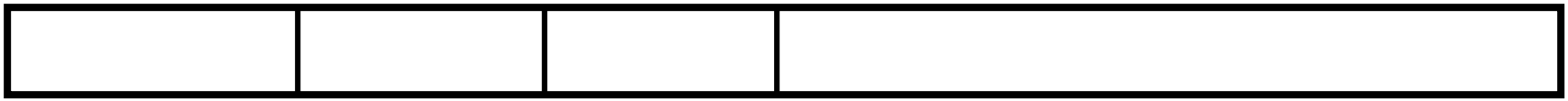
- MIPS instruction:

  lw $9, 12($8)

Field representation in decimal:



Field representation in binary:



Hexadecimal representation of instruction:

???

Try It Yourself #2

- MIPS instruction:

  lw $9, 12($8)

Field representation in decimal:

opcode	rs	rt	immediate
23_{16}	8	9	12

Field representation in binary:

100011	01001	01000	0000 0000 0000 1100

Hexadecimal representation of instruction:

8D09000C

Instruction Address

- As instructions are stored in memory, they too have addresses
  - **beq**, **bne**, **j** instructions use these addresses
- As instructions are 32-bit long, instruction addresses are word-aligned as well
- One register keeps address of instruction being executed: Program Counter (PC)

Branches: PC-Relative Addressing (1/5)

- Use I-Format

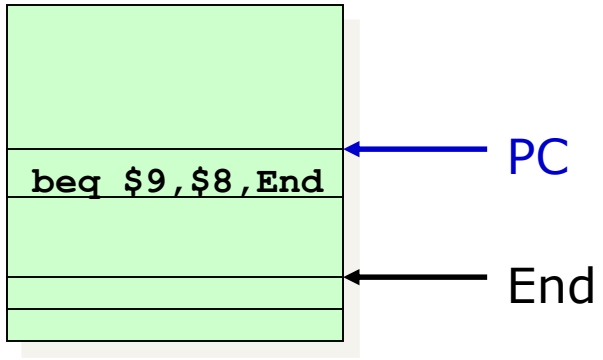
opcode	rs	rt	immediate

- **opcode** specifies **beq**, **bne**
- **rs** and **rt** specify registers to compare
- What can **immediate** specify?
  - **Immediate** is only 16 bits
  - Memory address is 32-bit
  - So **immediate** cannot specify entire address to branch to

Branches: PC-Relative Addressing (2/5)

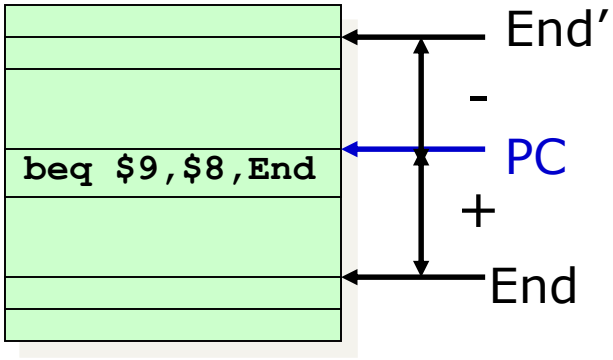
- How do we usually use branches?
  - Answer: **if-else, while, for**
  - Loops are generally small: typically up to 50 instructions
  - Unconditional jumps are done using jump instructions (j), not the branches

- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount



Branches: PC-Relative Addressing (3/5)

- Solution: specify target address relative to the PC
- Let the 16-bit **immediate** field be a signed two's complement integer to be added to the PC if we take the branch.
- Now we can branch ± 2^{15} bytes from the PC, which should be enough to cover almost any loop



Branches: PC-Relative Addressing (4/5)

- Instructions are word-aligned
  - number of bytes to add to the PC will always be a multiple of 4.
  - specify the **immediate** in words.
- Now we can branch $\pm 2^{15}$ words from the PC (or $\pm 2^{17}$ bytes)
- We can handle loops 4 times as large

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don't take the branch:

    PC = PC + 4

    PC+4 = address of next instruction
  - If we do take the branch:

    PC = (PC + 4) + (immediate × 4)
  - Observations
    - **immediate** field specifies the number of words to jump, which is simply the number of instructions to jump
    - **immediate** field can be positive or negative.
    - Due to hardware, add **immediate** to (PC+4), not to PC; will be clearer why later in the course

Branch: Example (1/3)

- MIPS Code:

```
Loop: beq  $9, $0, End
      add  $8, $8, $10
      addi $9, $9, -1
      j    Loop
End:
```

- **beq** branch is I-Format:

opcode	= 4 (look up in table)
rs	= 9 (first operand)
rt	= 0 (second operand)
immediate	= **???**

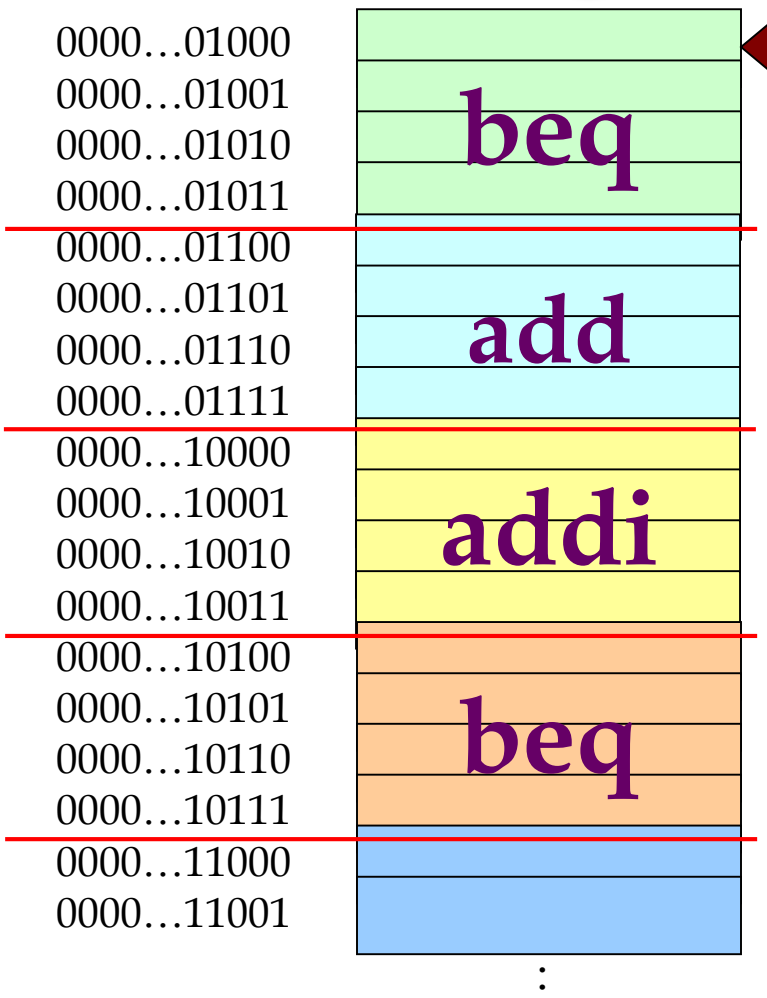
Branch: Example (2/3)

- MIPS Code:

```
Loop:  beq  $9, $0, End      # rel addr: 0
       add  $8, $8, $10      # rel addr: 4
       addi $9, $9, -1       # rel addr: 8
       j    Loop             # rel addr: 12
End:                         # rel addr: 16
```

- **immediate** field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch
  - In **beq** case, **immediate** = 3
  - End = (PC + 4) + (immediate × 4)

Addressing For Branch



PC

```
Loop:  beq  $9,$0,End      #Addr:  8
       add  $8,$8,$10      #Addr:12
       addi $9,$9,-1       #Addr:16
       beq  $0, $0, Loop   #Addr:20
End:                       #Addr:24
```

- PC = 8
- PC + 4 = 12
- Distance between End and (PC+4)
  = (24 – 12) bytes
  = 12 bytes
  = 3 words
- immediate = **3**
- new PC = (PC+4) + (immediatex4)
  = 12 + (3x4) = 24

Branch: Example (3/3)

- MIPS Code:

```
Loop: beq  $9, $0, End      # rel addr: 0
      add  $8, $8, $10      # rel addr: 4
      addi $9, $9, -1       # rel addr: 8
      j    Loop             # rel addr: 12
End:                        # rel addr: 16
```

Field representation in decimal:

opcode	rs	rt	immediate
4	9	0	3

Field representation in binary:

000100	01001	00000	0000000000000011

Try It Yourself #3

- MIPS Code:

```
Loop:   beq  $9, $0, End      # rel addr: 0
        add  $8, $8, $10      # rel addr: 4
        addi $9, $9, -1       # rel addr: 8
        beq  $0, $0, Loop     # rel addr: 12
End:                          # rel addr: 16
```

- What would be the **immediate** value for the second **beq** instruction?

Addressing For Branch


```
Loop:  beq  $9,$0,End      #Addr:  8
       add  $8,$8,$10      #Addr:12
       addi $9,$9,-1       #Addr:16
       beq  $0, $0, Loop   #Addr:20
End:                       #Addr:24
```

- PC = ?
- PC + 4 = ?
- Distance between Loop and (PC+4)
  =
  =
  =
- immediate =

J-Format (1/5)

- For branches, we assumed that we won't want to branch too far, so we can specify change in PC.
- For general jumps (j), we may jump to anywhere in memory.
- Ideally, we could specify a 32-bit memory address to jump to
- Unfortunately, we can't (why?)

J-Format (2/5)

- Define fields of the following number of bits each:

6 bits	26 bits

- As usual, each field has a name:

opcode	target address

- Keep **opcode** field identical to R-format and I-format for consistency
- Combine all other fields to make room for larger target address

J-Format (3/5)

- We can specify 26 bits of 32-bit address
- Optimization:
  - Just like with branches, jumps will only jump to word-aligned addresses, so last two bits are always 00 (in binary)
  - So let's just take this for granted and not even specify them
- Now we can specify 28 bits of 32-bit address

J-Format (4/5)

- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC
  - Technically, this means that we cannot jump to anywhere in memory, but it's adequate 99.9999…% of the time, since programs aren't that long
    - only if straddle a 256 MB boundary
- Special instruction if the program straddles 256MB boundary
  - Look up **jr** instruction if you are interested
  - Target address is specified through a register

J-Format (5/5)

- Summary:
  - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
  - { 4 bits , 26 bits , 2 bits } = 32 bit

Eg: {1010, 11111111111111111111111111, 00}
= 10101111111111111111111111111100

Assuming PC[31..28] = 1010
Target address = 11111111111111111111111111

Try It Yourself #4

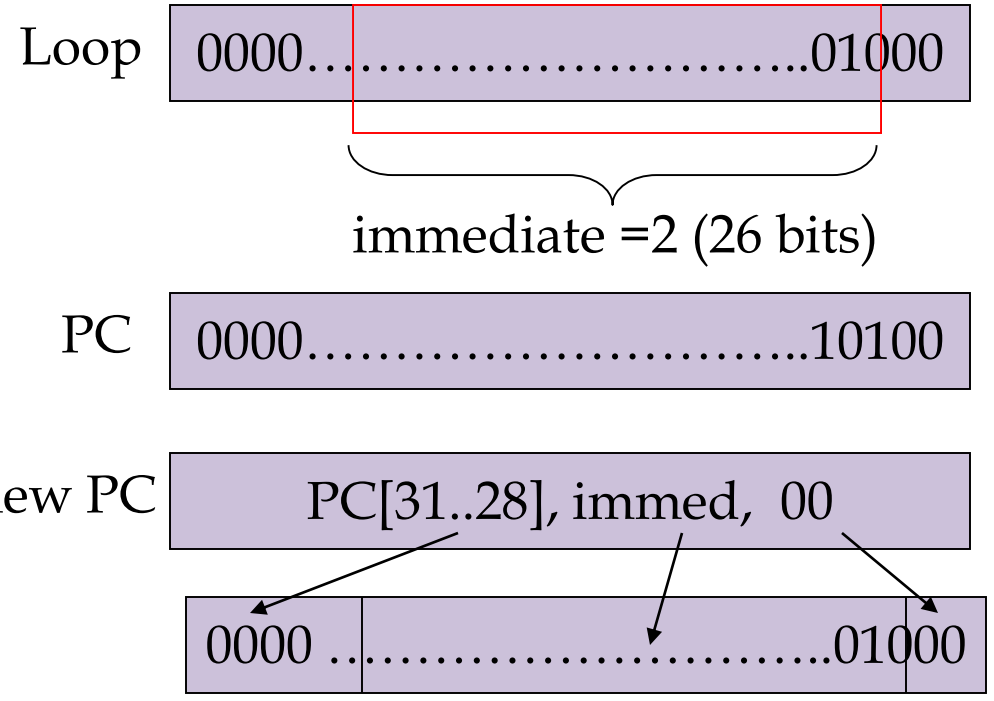
- MIPS Code:

```
Loop:   beq  $9, $0, End      # addr: 1000
        add  $8, $8, $10      # addr: 1004
        addi $9, $9, -1       # addr: 1008
        j    Loop             # addr: 1012
End:                          # addr: 1016
```

- What would be the **immediate** value for the **j Loop** instruction?

Addressing For JUMP (1/2)


```
Loop:  beq  $9,$0,End      #Addr:  8
       add  $8,$8,$10      #Addr:12
       addi $9,$9,-1       #Addr:16
       j    Loop           #Addr:20
End:                       #Addr:24
```



Addressing For JUMP (2/2)


```
Loop:  beq  $9,$0,End      #Addr:  8
       add  $8,$8,$10      #Addr:12
       addi $9,$9,-1       #Addr:16
       j    Far            #Addr:20
End:                       #Addr:24
```

Cannot jump to Far as PC[31..28] ≠Far[31..28]

Branching Far Away

- Given the instruction

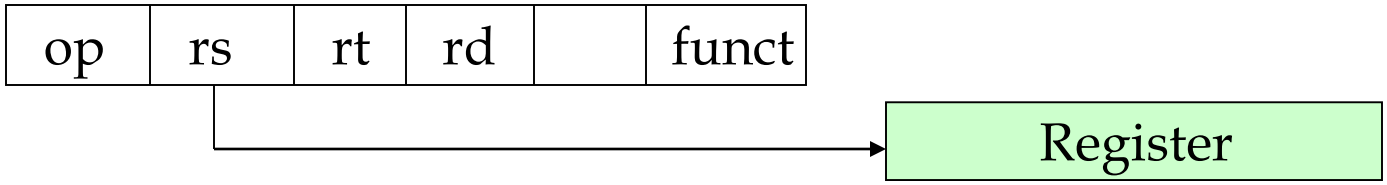
  beq $s0, $s1, L1

  Assume that the address **L1** is farther away from the PC than can be supported by **beq** and **bne** instructions

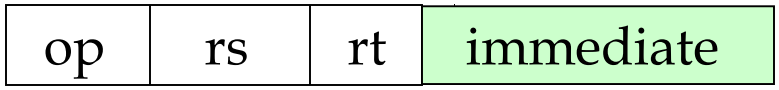
- Construct an equivalent code sequence with the help of unconditional (**j**) and conditional branch (**beq**, **bne**) instructions to accomplish this far away branching

Addressing Modes (1/3)

- **Register addressing**: operand is a register

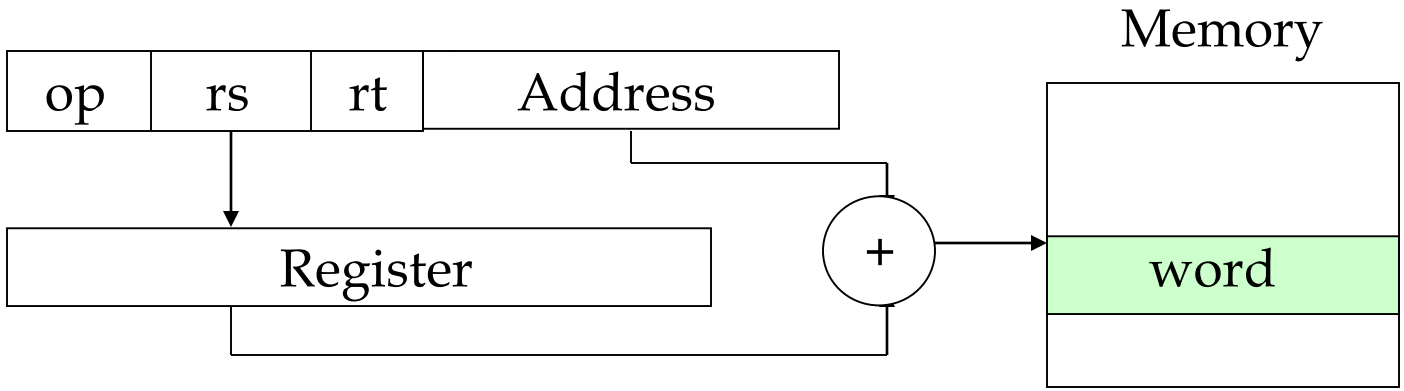


- **Immediate addressing**: operand is a constant within the instruction itself (**addi**, **andi**, **ori**, **slti**)



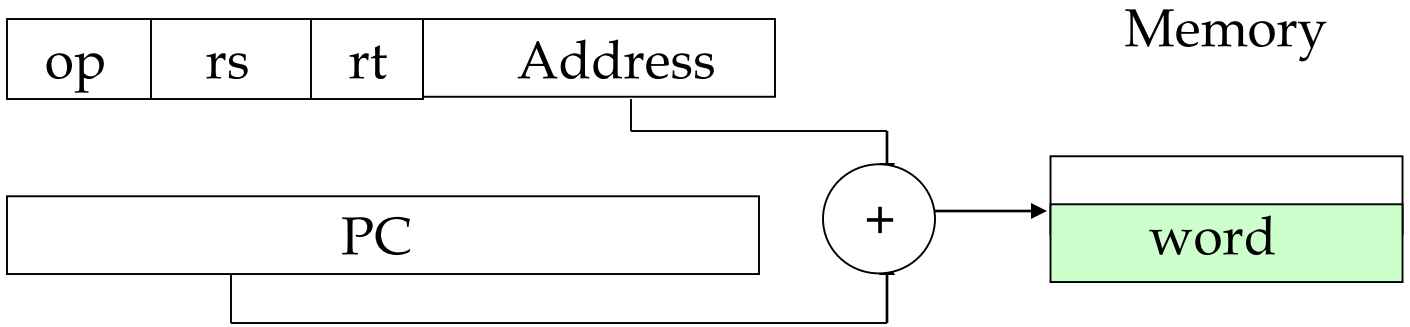
Addressing Modes (2/3)

- **Base addressing (displacement addressing)**: operand is at the memory location whose address is sum of a register and a constant in the instruction (**lw**, **sw**)

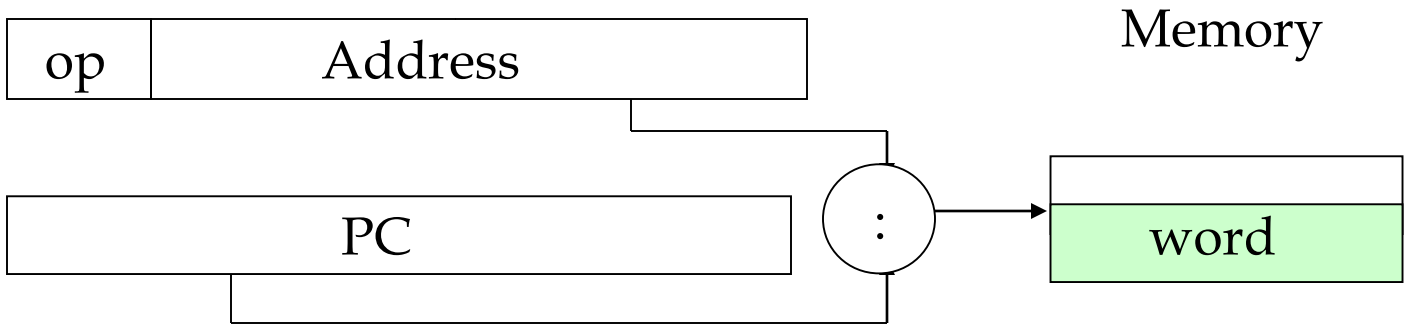


Addressing Modes (3/3)

- **PC-relative addressing**: address is sum of PC and constant in the instruction (**beq**, **bne**)



- **Pseudo-direct addressing**: 26-bit of instruction concatenated with upper 4-bits of PC (**j**)



SUMMARY (1/2)

- MIPS Machine Language Instruction:
  32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
- Branches use PC-relative addressing; jumps use pseudo-direct addressing
- Shifts use R-format, but other immediate instructions (addi, andi, ori) use I-format

SUMMARY (2/2)

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
Conditional branch	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

READING ASSIGNMENT

- Instructions: Language of the Computer
  - 4th edition
    - Section 2.5 Representing Instructions in the Computer
    - Section 2.10 MIPS Addressing for 32-Bit Immediates and Addresses