Topic 4

Instruction Coding

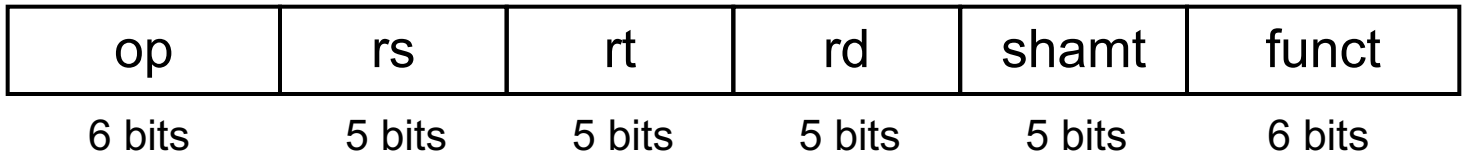
Representing Instructions

- Assembly instructions are translated into binary information
  - Called *machine code*
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Stored in 32-bit long memory locations
  - Small number of formats encode operation code (opcode), register numbers, …
  - **Regularity!**

Representing Instructions

- Three formats (types) to represent MIPS instructions
  - R-type (register)
  - I-type (immediate)
  - J-type (jump)

R-format



- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

Register Operands

- $zero: constant 0 (reg 0, also written as $0)
- $at: Assembler Temporary (reg 1, or $1)
- $v0, $v1: result values (reg's 2 and 3, or $2 and $3)
- $a0 – $a3: arguments (reg's 4 – 7, or $4 - $7)
- $t0 – $t7: temporaries (reg's 8 – 15, or $8 - $15)
- $s0 – $s7: saved (reg's 16 – 23, or $16 - $23)
- $t8, $t9: temporaries (reg's 24 and 25, or $24 and $25)
- $k0, $k1: reserved for OS kernel (reg's 26 and 27, $26/27)
- $gp: global pointer for static data (reg 28, or $28)
- $sp: stack pointer (reg 29, or $29)
- $fp: frame pointer (reg 30, or $30)
- $ra: return address (reg 31, or $31)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

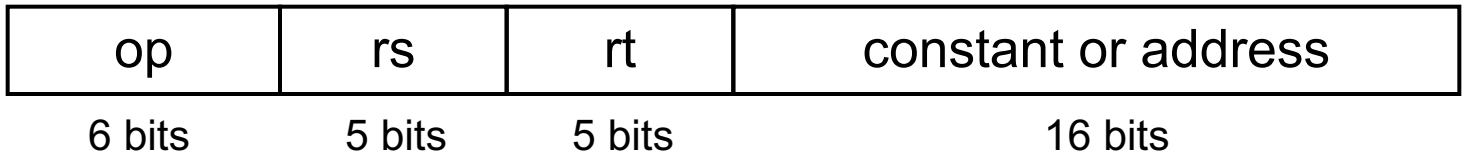
```
add $t0, $s1, $s2
```

Special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

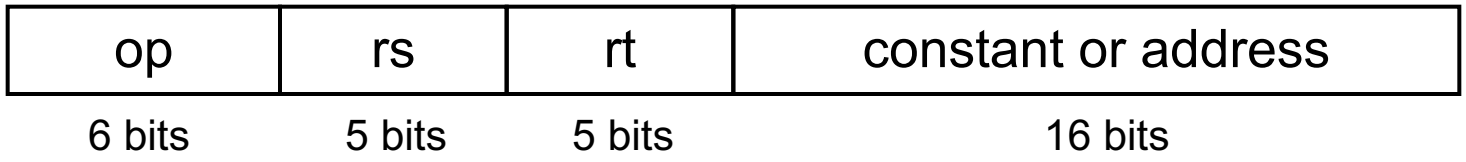
MIPS Reference Card

I-format



- 16-bit immediate number or address
  - rs: source or base address register
  - rt: destination or source register
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

I-format Example 1




```
addi $t0, $s0, 4
```

op	$s0	$t0	4
8	16	8	4
001000	10000	01000	0000000000000100

$$00100001000001000000000000000100_2 = 22080004_{16}$$

I-format Example 2

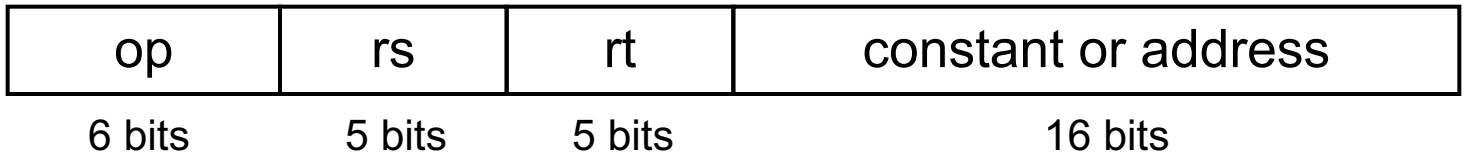
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

```
lw $t0, 4($s0)
```

op	$s0	$t0	4
23H	16	8	4
100011	10000	01000	0000000000000100

$$10001110000010000000000000000100_2 = 8E080004_{16}$$

I-format Example 3

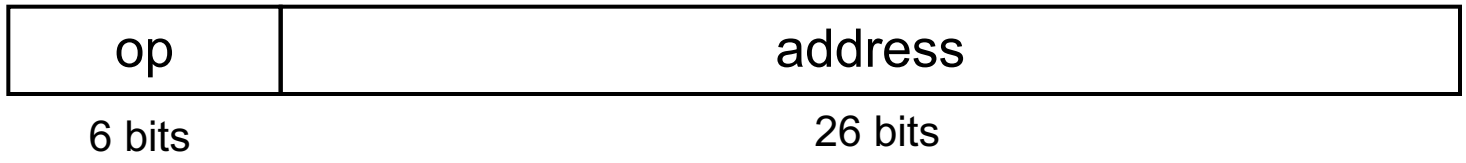



```
beq $s0, $t0, LOOP
```

op	$s0	$t0	Relative Address
4	16	8	Relative Address
000100	10000	01000	Relative Address

LOOP = PC+4+Relative Address*4

J-format



- Encode full address in instruction
- (Pseudo) Direct jump
  - Target address = PC[31:28] : (address × 4)

Target Addressing Example

- C code:

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

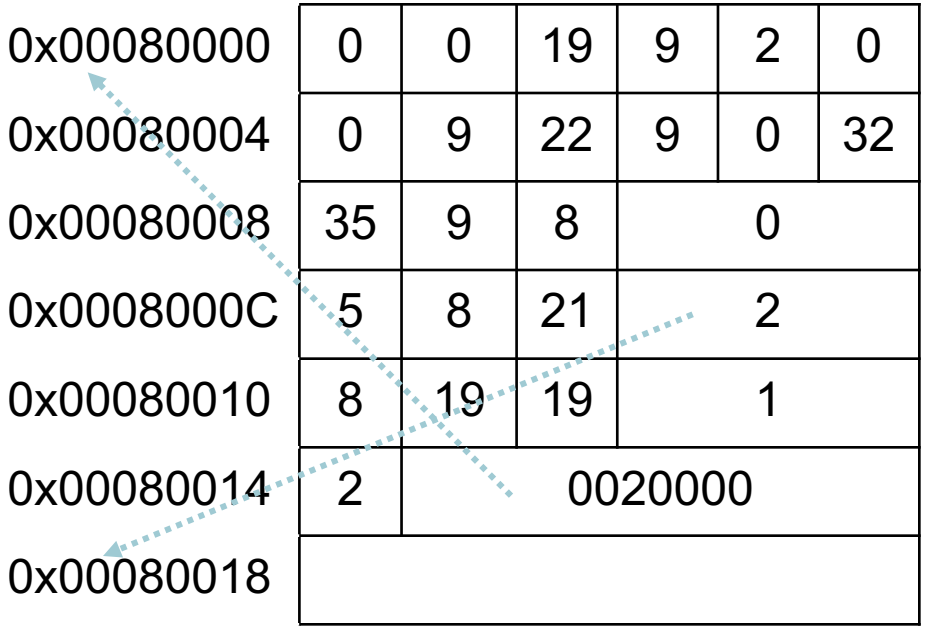
  - i in $s3, k in $s5, address of *save* in $s6
- Compiled MIPS code:

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

Target Addressing Example

- Assume Loop at location 00080000 (hex)

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



Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
  L2:   …
```

- May jump anywhere by **`jr`**

MIPS Addressing Mode

- How to get addresses?
  - Immediate Addressing
  - Register Addressing
  - Base Addressing
  - PC-relative addressing
  - Pseudodirect addressing

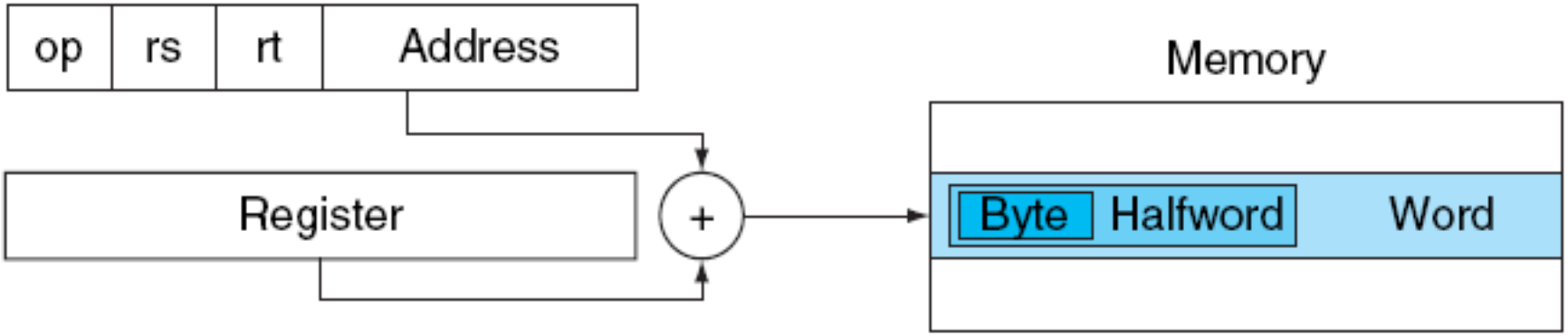
Immediate Addressing

- Operands are immediately provided in the instruction
- In I-type instructions
- Example
  - addi $t0, $s0, -1

Register Addressing

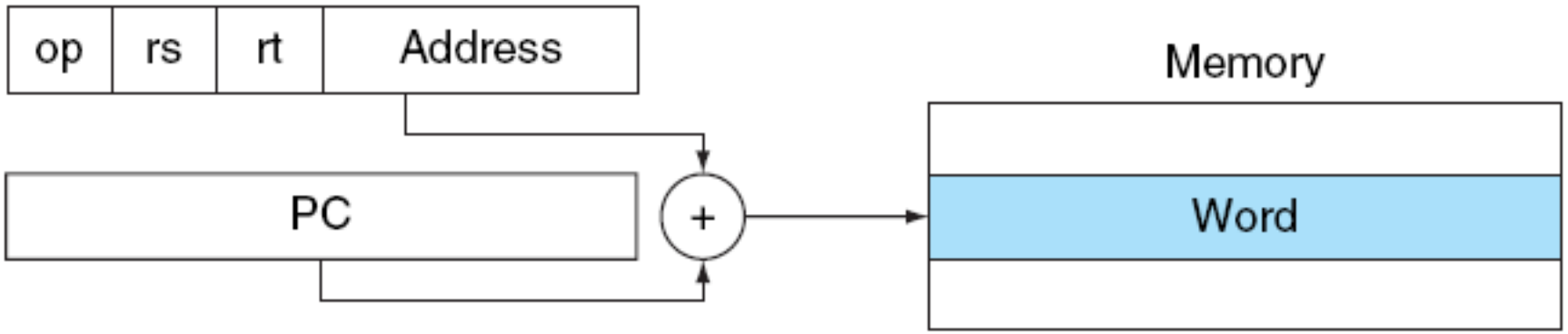
- All or some operands provided by register IDs directly
- Used in R-type and I-type instructions
- Example:
  - add $t0, $s0, $s1
  - beq $s0, $s1, FUNCTION

Base Addressing



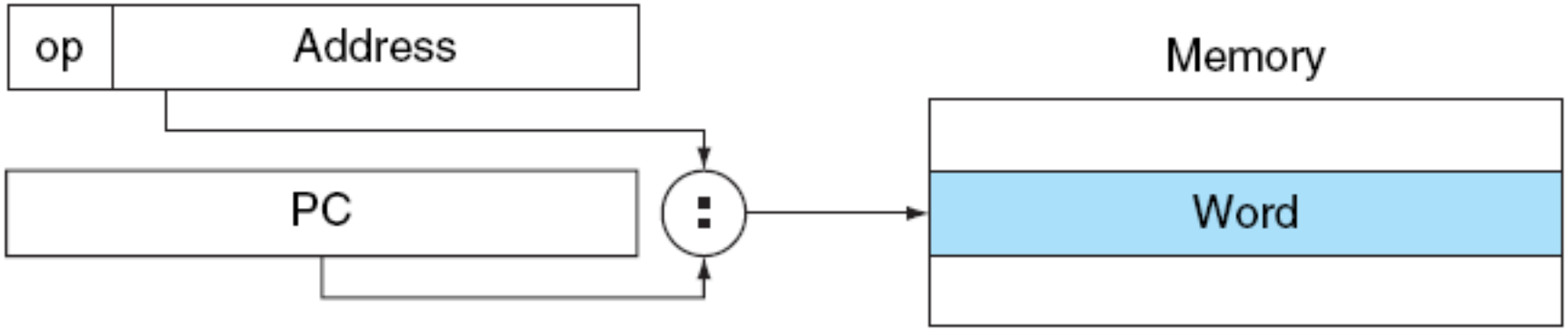
- Operands are provided by using base address of memory location
- Used in I-type
- Example
  - lw $t0, 32($s0)

PC-relative Addressing



- Operand relative to PC
- Used for near branch
  - Forward or backward
  - Target address = new PC + offset × 4
  - New PC = PC+4
- Example:
  - beq $s0, $s1, LESS (I-type)

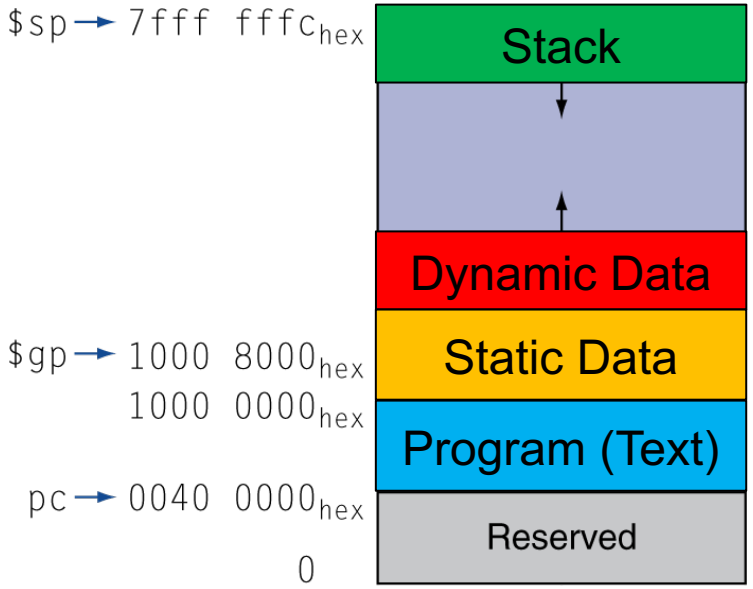
Pseudodirect Addressing



- Operand is a pseudodirect address of PC
  - Encode full address in instruction
- (Pseudo) Direct jump addressing
  - Target address = $PC_{31...28}$: (address $\times$ 4)
- Used in J-type instructions
  - j and jal (there is another jump: jr, R-type)

Big Picture – Recall

- Text: program code
  - PC initialized to 0x00400000
- Static data: global/static variables
  - $gp initialized to the middle of this segment, 0x10008000 allowing ±offset
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: storage for temporary variable in functions
  - $sp initialized to 0x7ffffffc, growing towards low address



Big Picture - Recall

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

- g, …, j in $a0, …, $a3, f in $s0, result in $v0

```
leaf_example:
  addi $sp, $sp, -4   #create spaces on stack
  sw   $s0, 0($sp)    #store data on stack
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)    #restore data from stack
  addi $sp, $sp, 4    #destroy spaces on stack
  jr   $ra            #return from function
```

Big Picture – Stored Program