EE108B – Lecture 3

MIPS Assembly Language II

Christos Kozyrakis

Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

- Urgent: sign up at EEclass and say if you are taking 3 or 4 units
- Homework #1 is available
  - Due on Tue 1/23, 5pm outside Gates 310
- Lab #1 is available
  - Due on Tuesday 1/30, midnight
  - Lab orientation after the class at Packard Hall
  -
- Programming assignment #1 will be available on Thursday
- EE108B Review Session
  - Friday: 2:15pm-3:05pm in Thornton 102

Review of MIPS Assembly Language I

- Instruction Set Architecture (ISA)
  - HW/SW interface
  - Multiple HW implementations of same interface

- Introduction to MIPS ISA and assembly programming
  - Register-register or load-store or RISC ISA

- Arithmetic (ALU) operations
  - Register & immediate operands
  - add, sub, addu, subu
  - addi, addiu

- Quick reminders
  - What is the difference between add and addu?
  - What is the difference between addi and addu?
  - Why don’t we have subi or subiu instructions?

Review of MIPS Assembly Language I

- Memory data transfer
  - displacement based addressing mode
  - lw, sw

Today’s Menu

- Finish Memory data transfer
  - Stores
  - Data alignment
  - Memory mapped I/O

- Control transfer instructions
  - Branches and jumps

- Machine language design
  - Binary encoding of assembly instructions
  - 3 MIPS instruction formats

Storing Data

- Storing data is just the reverse and the instruction is nearly identical
- Use the **`sw`** instruction to copy a word from the source register to an address in memory

```
sw src, offset(base)
```

- Offset value is signed

Storing Data Example

- Consider the example

```
*a = b + c;
```

- Use the `sw` instruction to store

```
add $t0, $s1, $s2    # $t0 = b + c
sw $t0, 0($s0)       # Memory[s0] = b + c
```

Storing to an Array

- Consider the example

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

- Use the sw instruction offset

```
add $t0, $s1, $s2          # $t0 = b + c
sw $t0, 12($s0)            # Memory[a[3]] = b + c
```

Complex Array Storage

- Consider the example

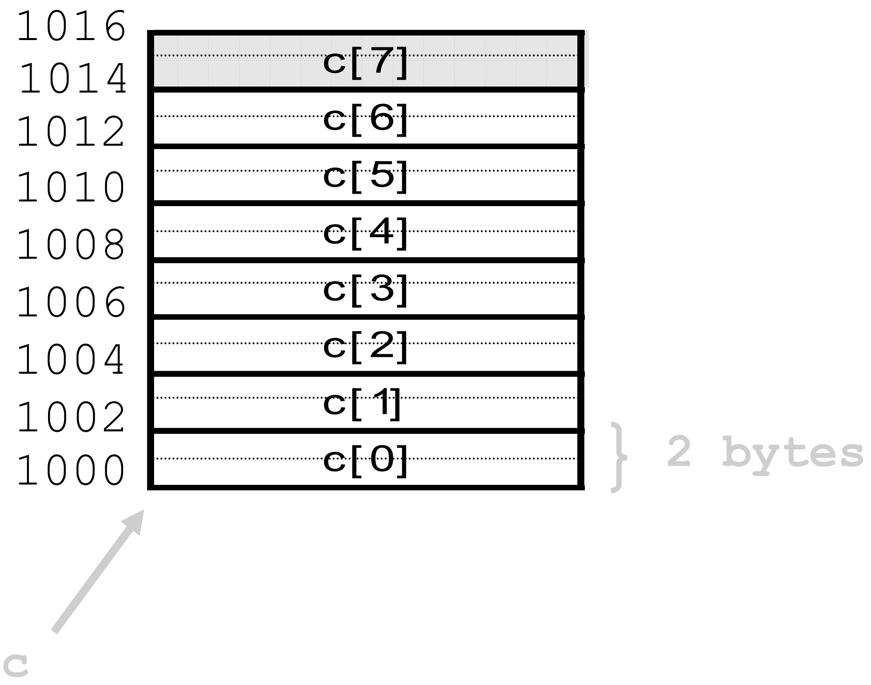
```
a[i] = b + c;
```

- Use the `sw` instruction offset

```
add $t0, $s1, $s2   # $t0 = b + c
sll $t1, $s3, 2     # $t1 = 4 * i
add $t2, $s0, $t1   # $t2 = a + 4*i
sw  $t0, 0($t2)     # Memory[a[i]] = b + c
```

A “short” Array Example

- ANSI C requires a short to be at least 16 bits and no longer than an int, but does not define the exact size
- For our purposes, treat a short as 2 bytes
- So, with a short array c[7] is at c + 7 * 2, shift left by 1



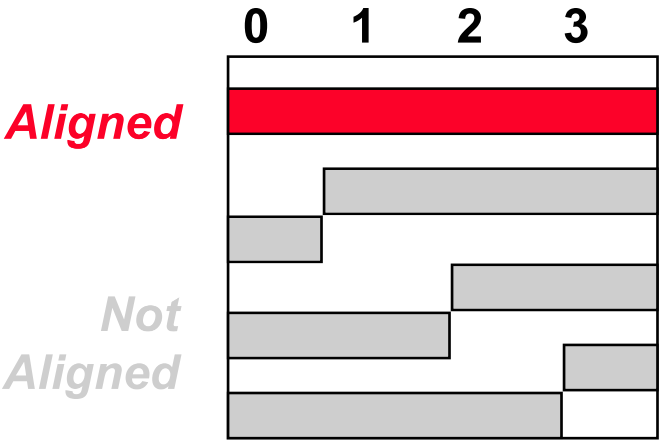
1016
1014 c[7]
1012 c[6]
1010 c[5]
1008 c[4]
1006 c[3]
1004 c[2]
1002 c[1]
1000 c[0]

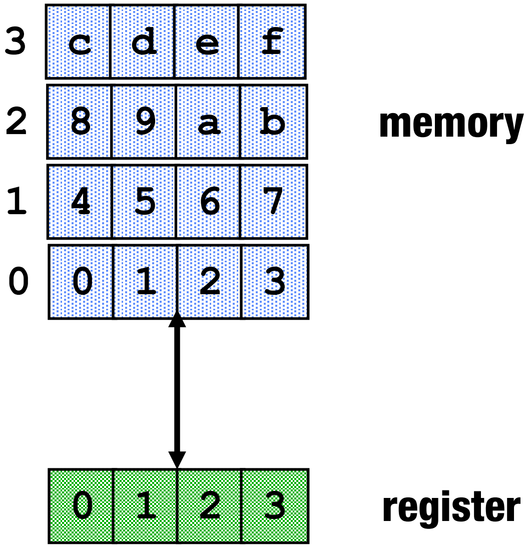
MIPS Integer Load/Store

Instruction	Example	Meaning	Comments
load word	lw $1, 8($2)	$1=Mem[8+$2]	Load word
load halfword	lh $1, 6($2)	$1=Mem[6+$2]	Load half; sign extend
load half unsign	lhu $1, 6($2)	$1=Mem[8+$2]	Load half; zero extend
load byte	lb $1, 5($2)	$1=Mem[5+$2]	Load byte; sign extend
load byte unsign	lbu $1, 5($2)	$1=Mem[5+$2]	Load byte; zero extend
store word	sw $1, 8($2)	Mem[8+$2]=$1	Store word
store half	sh $1, 6($2)	Mem[6+$2]=$1	Stores only lower 16 bits
store byte	sb $1, 5($2)	Mem[5+$2]=$1	Stores only lowest byte

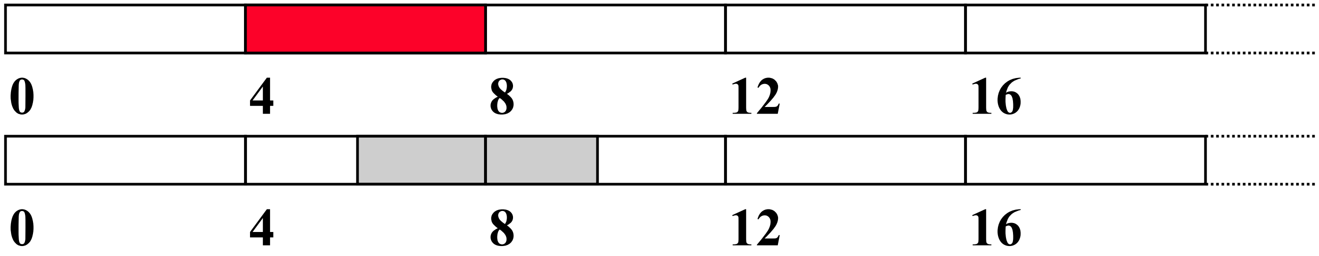
Alignment Restrictions

- In MIPS, data is required to fall on addresses that are even multiples of the data size





- Consider word (4 byte) memory access



Alignment Restrictions (cont)

- C Example

```
struct foo {
        char sm;        /*1 Byte*/
        short med;      /*2 Byte*/
        char sm1;       /*1 Byte*/
        int lrg;        /* 4 Bytes*/
}
```

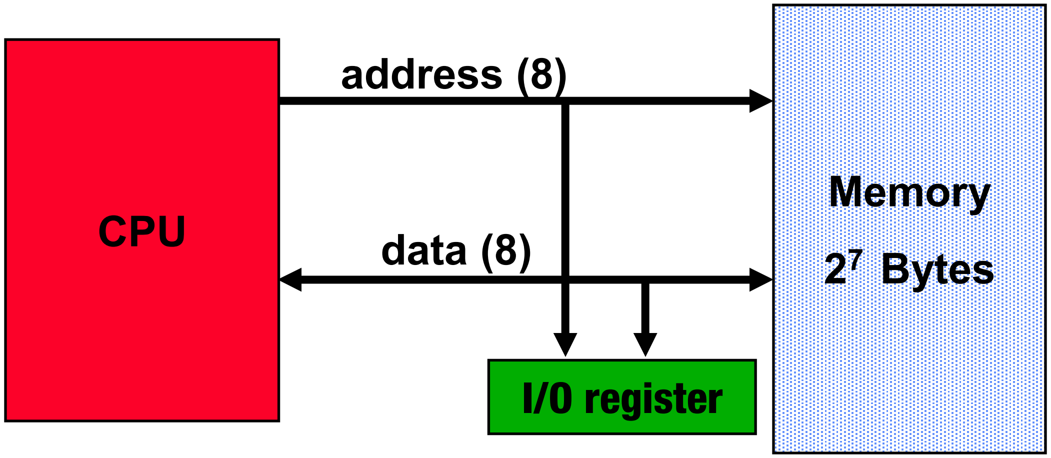
What is the size of this structure?

Byte offset	0	1	2 3	4	5 7	8 11
	sm	X	med	sm1	X	lrg

- Historically
  - Early machines (IBM 360 in 1964) required alignment
  - Removed in 1970s to reduce impact on programmers (e.g IBM 370, Intel x86)
  - Reintroduced by RISC to improve performance
- Also introduces challenges with memory organization with virtual memory, etc.

Memory Mapped I/O

- Data transfer instructions can be used to move data to and from I/O device registers
- A load operation moves data from a an I/O device register to a CPU register and a store operation moves data from a CPU register to a I/O device register



I/O register at address 0x80 (128)

Changing Control Flow

- One of the distinguishing characteristics of computers is the ability to evaluate conditions and change control flow
  - If-then-else
  - Loops
  - Case statements

- Control flow instructions
  - Conditional branch instructions are known as branches
  - Unconditional changes in the control flow are called jumps
- The target of the branch/jump is a label

Conditional: Equality

- The simplest conditional test is the `beq` instruction for equality

```
      beq reg1, reg2, label
```

- Consider the code

```
            if (a == b) goto L1;
            // Do something
      L1:   // Continue
```

- Use the `beq` instruction

```
            beq $s0, $s1, L1
            # Do something
            # . . .
      L1:   # Continue
```

Conditional: Not Equal

- The bne instruction for not equal

```
    bne reg1, reg2, label
```

- Consider the code

```
          if (a != b) goto L1;
          // Do something
    L1:   // Continue
```

- Use the bne instruction

```
          bne $s0, $s1, L1
          # Do something
    L1:   # Continue
```

Unconditional: Jumps

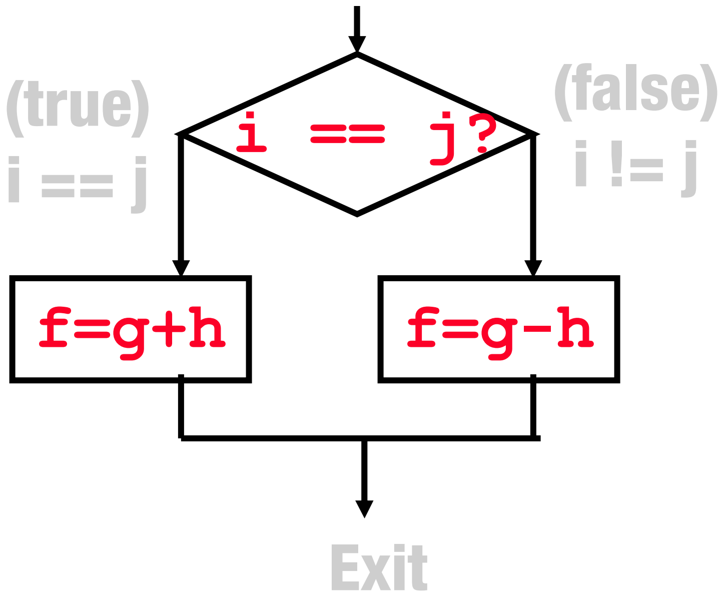
- The `j` instruction jumps to a label

```
    j label
```

If-then-else Example

- Consider the code

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



If-then-else Solution

- Create labels and use equality instruction

```
       beq $s3, $s4, True   # Branch if i == j
False: subu $s0, $s1, $s2   # f = g - h
       j Exit               # Go to Exit
True:  add $s0, $s1, $s2    # f = g + h
Exit:
```

Other Comparisons

- Other conditional arithmetic operators are useful in evaluating conditional expressions using <, >, <=, >=

- Use compare instruction to "set" register to 1 when condition met

- Consider the following C code

```
    if (f < g) goto Less;
```

- Solution

```
    slt $t0, $s0, $s1              # $t0 = 1 if $s0 < $s1
    bne $t0, $zero, Less           # Goto Less if $t0 != 0
```

MIPS Comparisons

Instruction	Example	Meaning	Comments
set less than	`slt $1, $2, $3`	$1 = ($2 < $3)	comp less than signed
set less than imm	`slti $1, $2, 100`	$1 = ($2 < 100)	comp w/const signed
set less than uns	`sltu $1, $2, $3`	$1 = ($2 < $3)	comp < unsigned
set l.t. imm. uns	`sltiu $1, $2, 100`	$1 = ($2 < 100)	comp < const unsigned

- C

```
if (a < 8)
```

- Assembly

```
slti $v0,$a0,8              # $v0 = a < 8
beq  $v0,$zero, Exceed      # goto Exceed if $v0 == 0
```

C Example

```c
int sum_pow2(int b, int c)
{
    int pow2 [8] = {1, 2, 4, 8, 16, 32, 64, 128};
    int  a, ret;
    a = b + c;
    if (a < 8)
        ret = pow2[a];
    else
        ret = 0;
    return(ret);
}
```

```
sum_pow2:                                  # $a0 = b, $a1 = c
        addu  $a0,$a0,$a1                  # a = b + c, $a0 = a
        slti  $v0,$a0,8                    # $v0 = a < 8
        beq   $v0,$zero, Exceed            # goto Exceed if $v0 == 0
        addiu $v1,$sp,8                    # $v1 = pow2 address
        sll   $v0,$a0,2                    # $v0 = a*4
        addu  $v0,$v0,$v1                  # $v0 = pow2 + a*4
        lw    $v0,0($v0)                   # $v0 = pow2[a]
        j     Return                       # goto Return
Exceed: addu  $v0,$zero,$zero              # $v0 = 0

Return: jr    ra                           # return sum_pow2
```

C Example Revised

```c
int sum_pow2(int b, int c)
{
    int pow2 [8] = {1, 2, 4, 8, 16, 32, 64, 128};
    int  a, ret;
    a = b + c;
    if (a >= 0 && a < 8)
        ret = pow2[a];
    else
        ret = 0;
    return(ret);
}
```

```
sum_pow2:                                     # $a0 = b, $a1 = c
         addu  $a0,$a0,$a1                    # a = b + c, $a0 = a
         bltz  $a0, Exceed                    # goto Exceed if $v0 < 0
         slti  $v0,$a0,8                      # $v0 = a < 8
         beq   $v0,$zero, Exceed              # goto Exceed if $v0 == 0
         addiu $v1,$sp,8                      # $v1 = pow2 address
         sll   $v0,$a0,2                      # $v0 = a*4
         addu  $v0,$v0,$v1                    # $v0 = pow2 + a*4
         lw    $v0,0($v0)                     # $v0 = pow2[a]
         j     Return                         # goto Return
Exceed:  addu  $v0,$zero,$zero                # $v0 = 0

Return:  jr    ra                             # return sum_pow2
```

MIPS Jumps & Branches

Instruction	Example	Meaning
jump	j L	goto L
jump register	jr $1	goto value in $1
jump and link	jal L	goto L and set $ra
jump and link register	jalr $1	goto $1 and set $ra
branch equal	beq $1, $2, L	if ($1 == $s2) goto L
branch not eq	bne $1, $2, L	if ($1 != $2) goto L
branch l.t. 0	bltz $1, L	if ($1 < 0) goto L
branch l.t./eq 0	blez $1, L	if ($1 <= 0) goto L
branch g.t. 0	bgtz $1, L	if ($1 > 0) goto L
branch g.t./eq 0	bgez $1, L	if ($1 >= 0) goto L

Support for Simple Branches Only

- Notice that there is no branch less than instruction for comparing two registers?
  - The reason is that such an instruction would be too complicated and might require a longer clock cycle time
  - Therefore, conditionals that do not compare against zero take at least two instructions where the first is a set and the second is a conditional branch

- As we'll see later, this is a design trade-off
  - Less time per instruction Vs. fewer instructions
    - How do you decide what to do?
  - Other RISC ISAs made a different choice (e.g. HP's PA-RISC)

While loop in C

- Consider a `while`loop

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

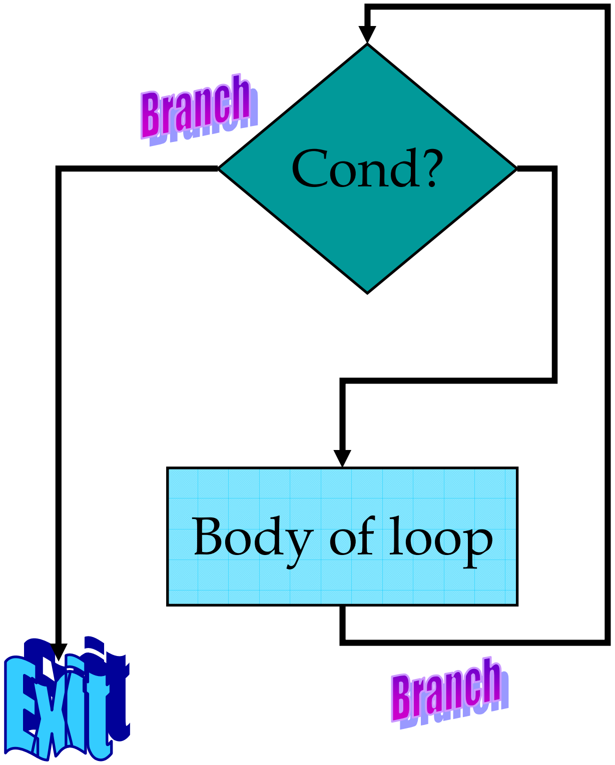
- Assembly loop
- Assume `i = $s0, j = $s1, k = $s2`

```
Loop:   sll $t0, $s0, 2             # $t0 = 4 * i
        addu $t1, $t0, $s3          # $t1 = &(A[i])
        lw $t2, 0($t1)              # $t2 = A[i]
        bne $t2, $s2, Exit          # goto Exit if !=
        addu $s0, $s0, $s1          # i = i + j
        j Loop                      # goto Loop
Exit:
```

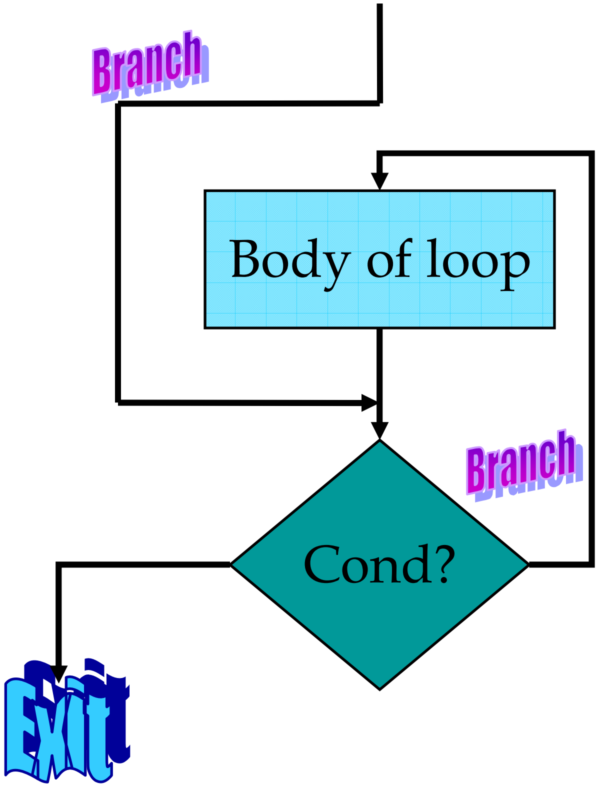
- Basic block :
  - Maximal sequence of instructions with out branches or branch targets

Improve Loop Efficiency

- Code uses two branches/iteration:



- Better structure:



Improved Loop Solution

- Remove extra jump from loop body

```
        j Cond                  # goto Cond
Loop:   addu $s0, $s0, $s1      # i = i + j
Cond:   sll $t0, $s0, 2         # $t0 = 4 * i
        addu $t1, $t0, $s3      # $t1 = &(A[i])
        lw $t2, 0($t1)          # $t2 = A[i]
        beq $t2, $s2, Loop      # goto Loop if ==
Exit:
```

- Reduced loop from 6 to 5 instructions
  - Even small improvements important if loop executes many times

Machine Language Representation

- Instructions are represented as binary data in memory
    - “Stored program” - Von Neumann
  - Simplicity
    - One memory system
    - Same addresses used for branches, procedures, data, etc.
  - The only difference is how bits are interpreted
    - What are the risks of this decision?

- Binary compatibility (backwards)
  - Commercial software relies on ability to work on next generation hardware
  - This leads to a very long life for an ISA

MIPS Instruction Encoding

- MIPS Instructions are encoded in different forms, depending upon the arguments
  - R-format, I-format, J-format
- MIPS architecture has three instruction formats, all 32 bits in length
  - Regularity is simpler and improves performance
- A 6 bit opcode appears at the beginning of each instruction
  - Control logic based on decoded instruction type
- See “green card” and Page 103 for a list

R-Format

- Used by ALU instructions
- Uses three registers: one for destination and two for source

Bits	6	5	5	5	5	6
	OP=0	rs	rt	rd	sa	funct
		First Source Register	Second Source Register	Result Register	Shift Amount	Function Code

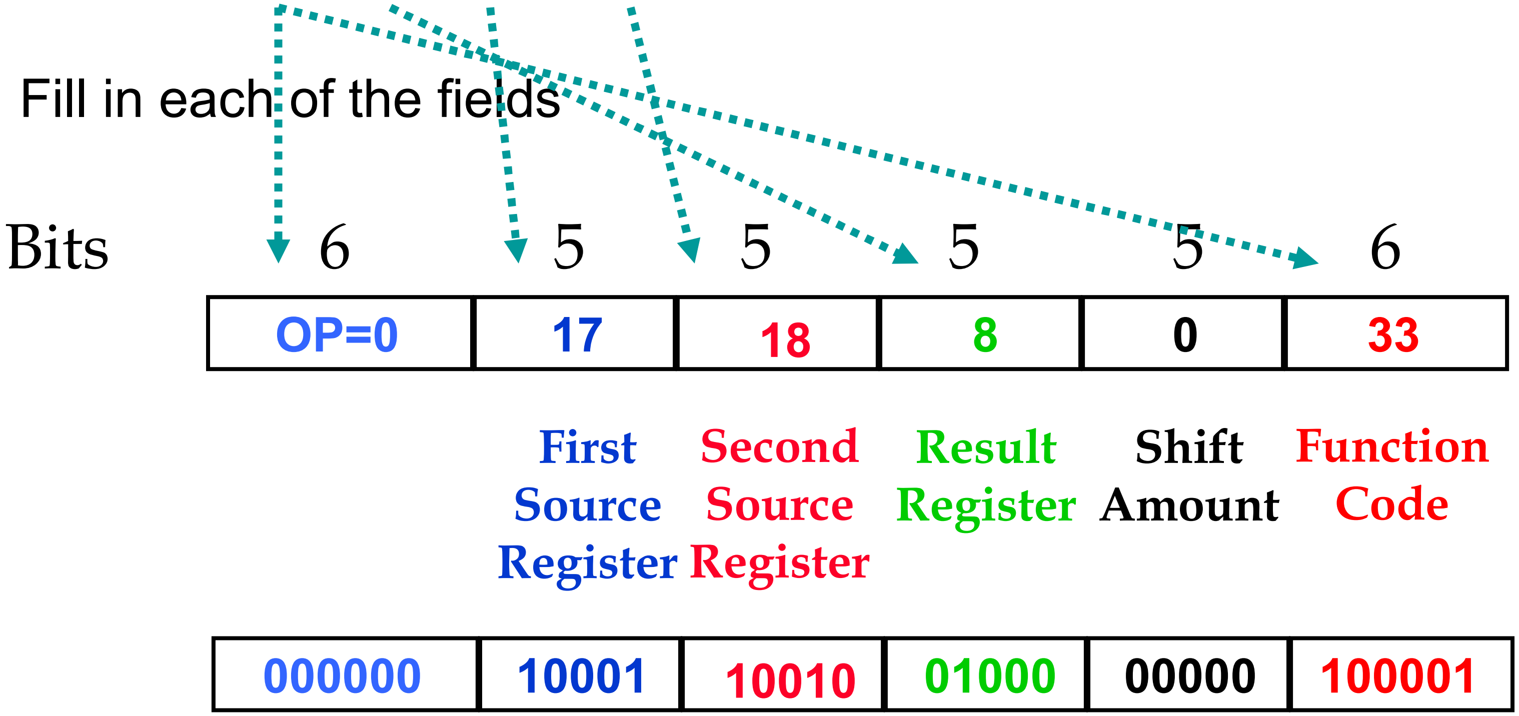
- Function code specifies which operation

R-Format Example

- Consider the `addu` instruction

```
addu $t0, $s1, $s2
```

- Fill in each of the fields



Bits	6	5	5	5	5	6
	OP=0	17	18	8	0	33
		First Source Register	Second Source Register	Result Register	Shift Amount	Function Code
	000000	10001	10010	01000	00000	100001

R-Format Limitations

- The R-Format works well for ALU-type operations, but does not work well for some of the other instructions types

- Consider for example the lw instruction that takes an offset in addition to two registers
  - R-format would provide 5 bits for the offset
  - Offsets of only 32 are not all that useful!

I-Format

- The immediate instruction format
  - Uses different opcodes for each instruction
  - Immediate field is signed (positive/negative constants)
  - Used for loads and stores as well as other instructions with immediates (addi, lui, etc.)
  - Also used for branches

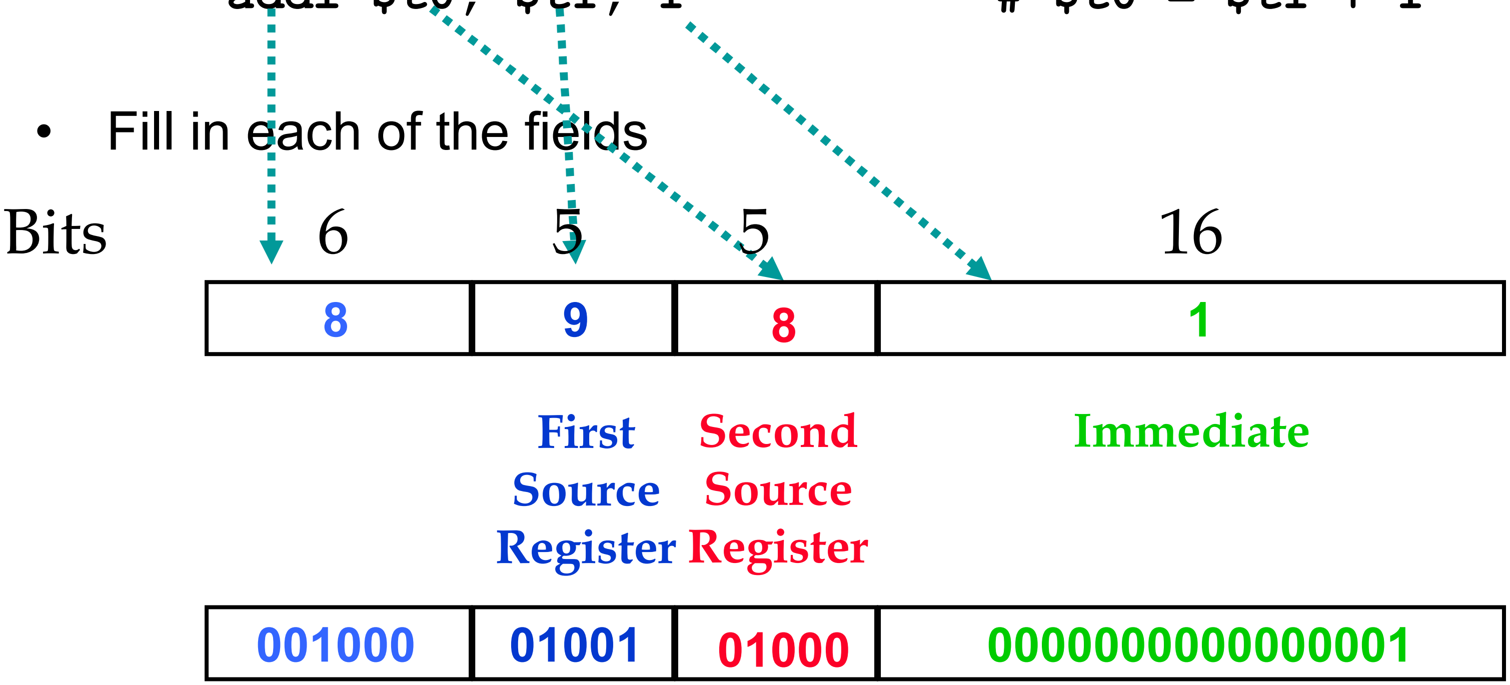
Bits	6	5	5	16
	OP	rs	rt	imm
		First Source Register	Second Source Register or dest	Immediate

I-Format Example

- Consider the `addi` instruction

```
addi $t0, $t1, 1                # $t0 = $t1 + 1
```

- Fill in each of the fields



Bits	6	5	5	16
	8	9	8	1
		First Source Register	Second Source Register	Immediate
	001000	01001	01000	0000000000000001

Another I-Format Example

- Consider the while loop

```
Loop:  addu $t0, $s0, $s0          # $t0 = 2 * i
       addu $t0, $t0, $t0          # $t0 = 4 * i
       add $t1, $t0, $s3           # $t1 = &(A[i])
       lw $t2, 0($t1)              # $t2 = A[i]
       bne $t2, $s2, Exit          # goto Exit if !=
       addu $s0, $s0, $s1          # i = i + j
       j Loop                      # goto Loop
Exit:
```

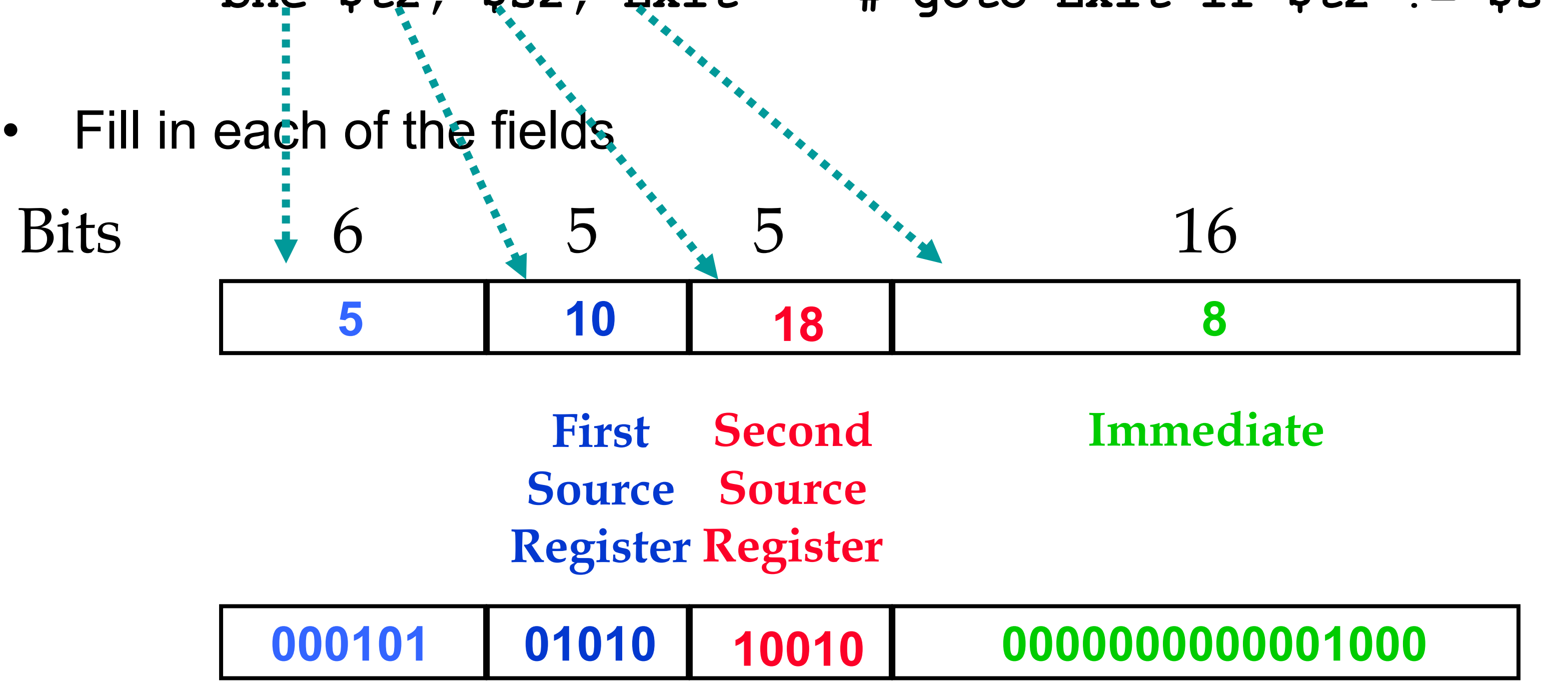
- Pretend the first instruction is located at address 80000

I-Format Example (Incorrect)

- Consider the `bne` instruction

```
bne $t2, $s2, Exit     # goto Exit if $t2 != $s2
```

- Fill in each of the fields



Bits	6	5	5	16
	5	10	18	8
		First Source Register	Second Source Register	Immediate
	000101	01010	10010	0000000000001000

- This is not the optimum encoding

PC Relative Addressing

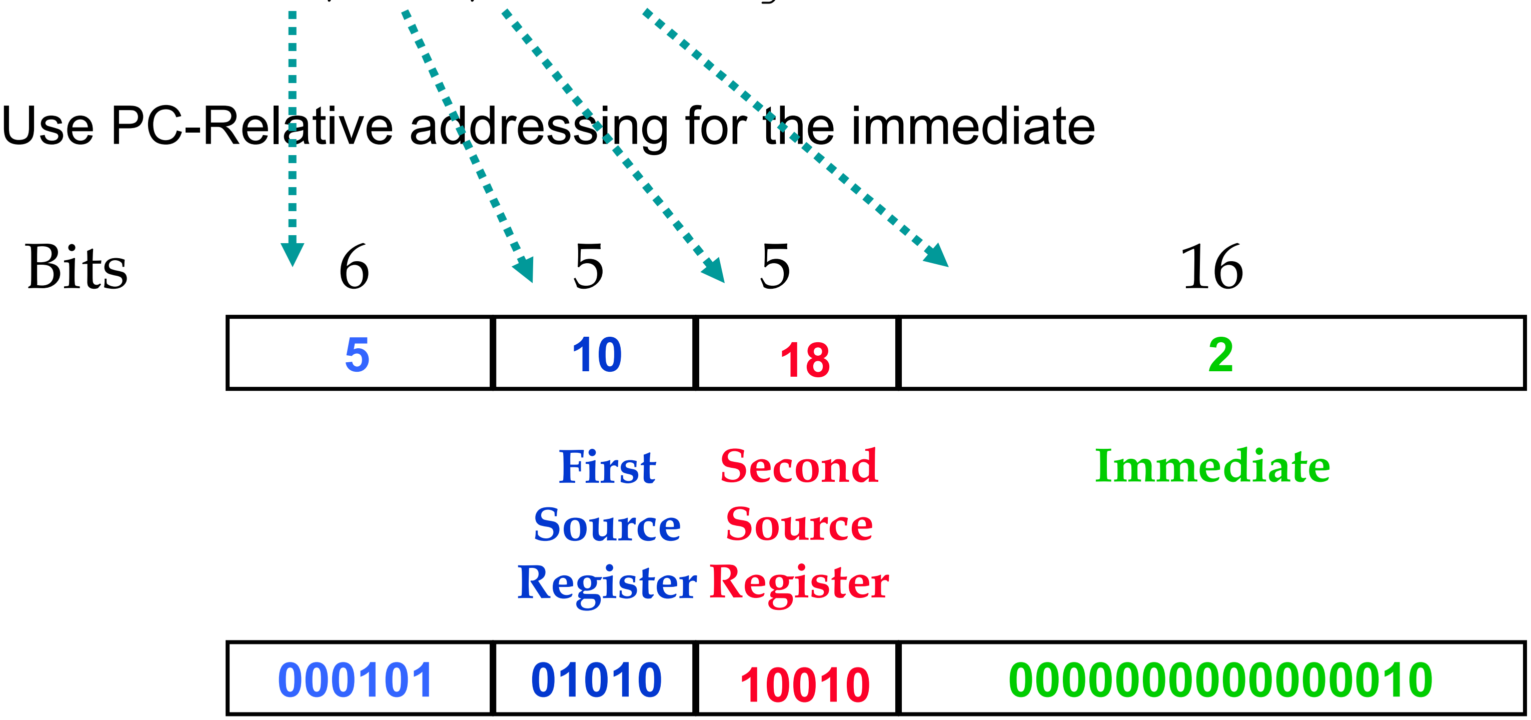
- How can we improve our use of immediate addresses when branching?
- Since instructions are always 32 bits long and word addressing requires alignment, every address must be a multiple of 4 bytes
- Therefore, we actually branch to the address that is PC + 4 + 4 × immediate

I-Format Example

- Re-consider the `bne` instruction

```
bne $t2, $s2, Exit  # goto Exit if $t2 != $s2
```

- Use PC-Relative addressing for the immediate



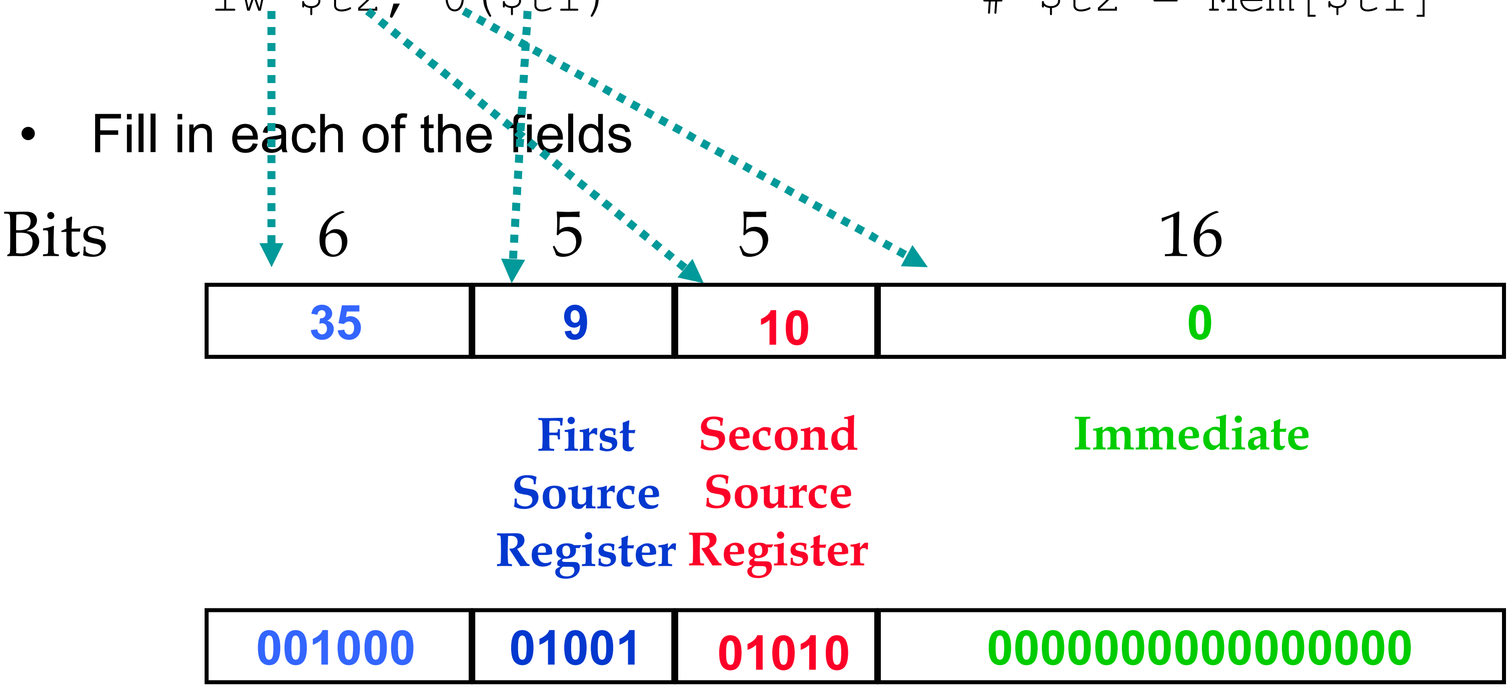
Bits	6	5	5	16
	5	10	18	2
		First Source Register	Second Source Register	Immediate
	000101	01010	10010	0000000000000010

I-Format Example: Load/Store

- Consider the `lw` instruction

```
lw $t2, 0($t1)                # $t2 = Mem[$t1]
```

- Fill in each of the fields



Bits	6	5	5	16
	35	9	10	0
		First Source Register	Second Source Register	Immediate
	001000	01001	01010	0000000000000000

Branching Far Away

- If the target is greater than -2^{15} to 2^{15}-1 words away, then the compiler inverts the condition and inserts an unconditional jump
- Consider the example where L1 is far away

```
beq $s0, $s1, L1            # goto L1 if S$0=$s1
```

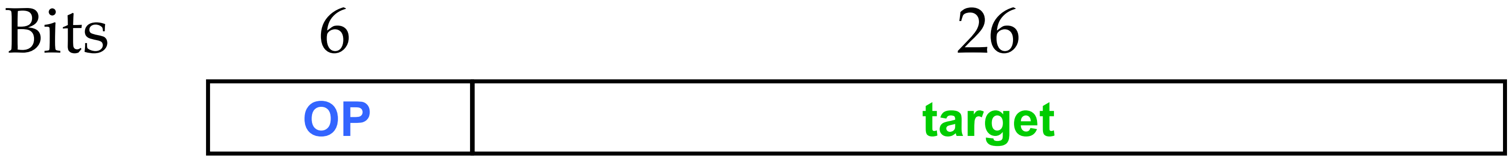
- Can be rewritten as

```
        bne $s0, $s1, L2    # Inverted
        j L1                # Unconditional jump
L2:
```

- Compiler must be careful not to cross 256 MB boundaries with jump instructions

J-Format

- The jump instruction format
  - Different opcodes for each instruction
  - Examples include j and jal instructions
  - Absolute addressing since long jumps are common
  - Based on word addressing (target × 4)
  - Pseudodirect addressing where 28 bits from target, and remaining 4 bits come from upper bits of PC



Jump Target Address

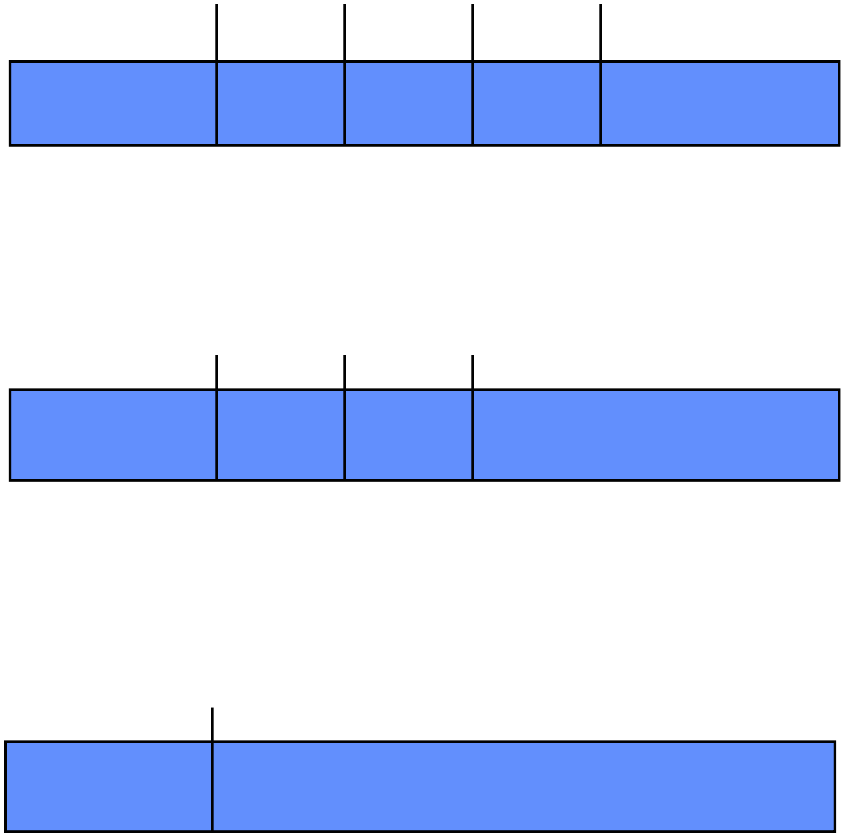
$$\text{Jump PC} = PC_{31..28} || \text{target} || 00$$

Complete Example

Now we can write the complete example for our while loop

80000	0	16	16	8	0	33
80004	0	8	8	8	0	33
80008	0	8	19	9	0	33
80012	35	9	10	0		
80016	5	10	18	2		
80020	0	16	17	16	0	33
80024	2	20000				
80028	…					

MIPS Machine Instruction Review: Instruction Format Summary



```
sum_pow2:                               # $a0 = b, $a1 = c
          addu  $a0,$a0,$a1             # a = b + c, $a0 = a
          bltz  $a0, Exceed             # goto Exceed if $a0 < 0
          slti  $v0,$a0,8               # $v0 = a < 8
          beq   $v0,$zero, Exceed       # goto Exceed if $v0 == 0
          addiu $v1,$sp,8               # $v1 = pow2 address
          sll   $v0,$a0,2               # $v0 = a*4
          addu  $v0,$v0,$v1             # $v0 = pow2 + a*4
          lw    $v0,0($v0)              # $v0 = pow2[a]
          b     Return                  # goto Return
Exceed:   addu  $v0,zero,zero           # $v0 = 0

Return:   jr    ra                      # return sum_pow2
```

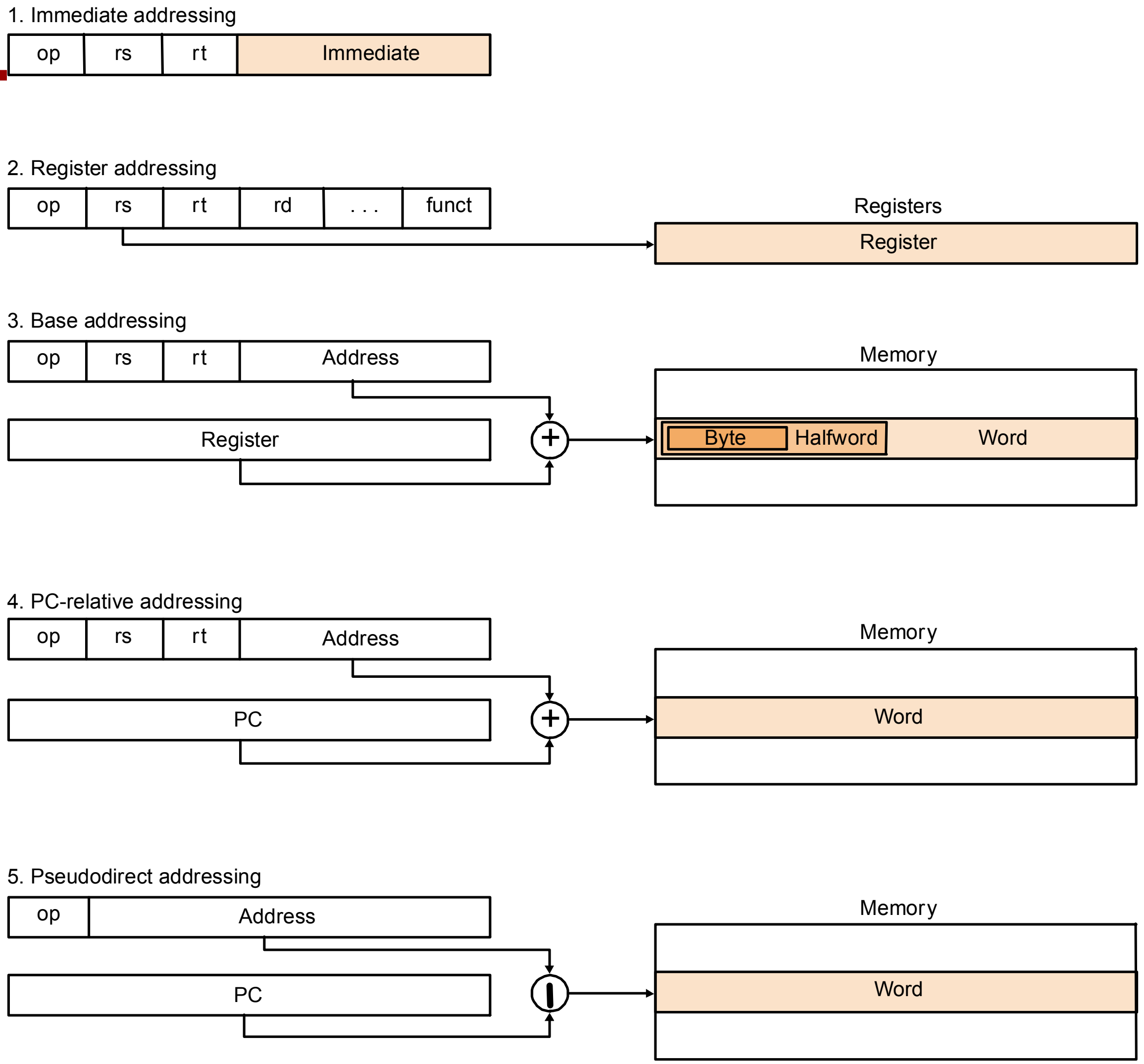
sum_pow2 Revised Machine and DisAssembly

```
sum_pow2:
0x400a98:  00 85 20 21  addu a0,a0,a1
0x400a9c:  04 80 00 08  bltz a0,0x400abc
0x400aa0:  28 82 00 06  slti v0,a0,6
0x400aa4:  10 40 00 06  beq v0,zero,0x400abc
0x400aa8:  27 a3 00 08  addiu v1,sp,8
0x400aac:  00 04 10 80  sll v0,a0,2
0x400ab0:  00 43 10 21  addu v0,v0,v1
0x400ab4:  8c 42 00 00  lw v0,0(v0)
0x400ab8:  10 00 00 01  j 0x400ac0
0x400abc:  00 00 10 21  addu v0,zero,zero
0x400ac0:  03 e0 00 08  jr ra
```

Addressing Modes Summary

- Register addressing
  - Operand is a register (e.g. ALU)
- Base/displacement addressing (ex. load/store)
  - Operand is at the memory location that is the sum of
  - a base register + a constant
- Immediate addressing (e.g. constants)
  - Operand is a constant within the instruction itself
- PC-relative addressing (e.g. branch)
  - Address is the sum of PC and constant in instruction (e.g. branch)
- Pseudo-direct addressing (e.g. jump)
  - Target address is concatenation of field in instruction and the PC

Addressing Modes Summary



1. Immediate addressing

op | rs | rt | Immediate

2. Register addressing

op | rs | rt | rd | . . . | funct

Registers

Register

3. Base addressing

op | rs | rt | Address

Register

Memory

Byte Halfword Word

4. PC-relative addressing

op | rs | rt | Address

PC

Memory

Word

5. Pseudodirect addressing

op | Address

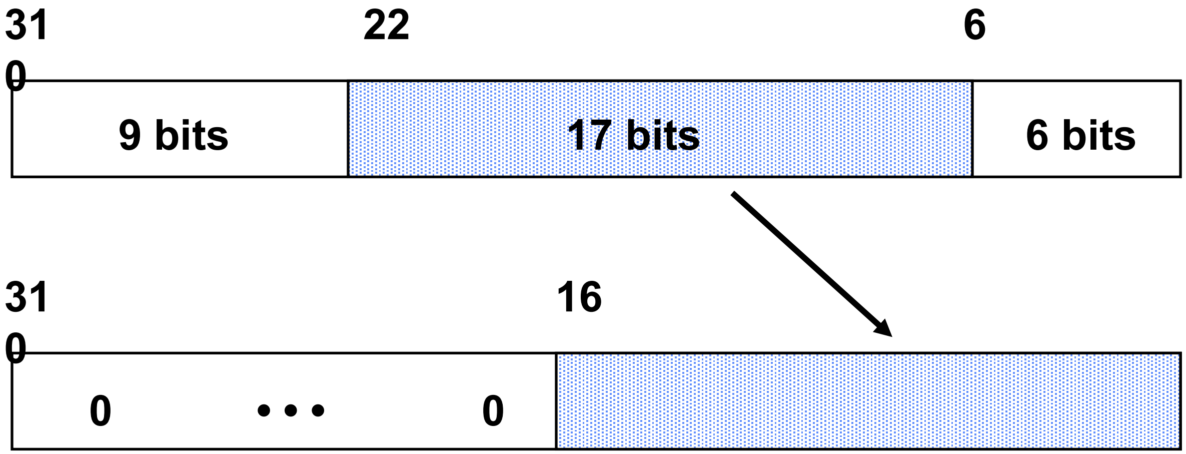
PC

Memory

Word

Logical Operators

- Bitwise operators often useful for bit manipulation



```
sll $t0, $t3, 9  # shift $t3 left by 9, store in $t0
srl $t0, $t0, 15 # shift $t0 right by 15
```

- Always operate unsigned except for arithmetic shifts

MIPS Logical Instructions

Instruction	Example	Meaning	Comments
and	and $1, $2, $3	$1 = $2 & $3	Logical AND
or	or $1, $2, $3	$1 = $2 | $3	Logical OR
xor	xor $1, $2, $3	$1 = $2 □.$3	Logical XOR
nor	nor $1, $2, $3	$1 = ~($2 | $3)	Logical NOR
and immediate	andi $1, $2, 10	$1 = $2 & 10	Logical AND w. constant
or immediate	ori $1, $2, 10	$1 = $2 | 10	Logical OR w. constant
xor immediate	xori $1, $2, 10	$1 = ~$2 & ~10	Logical XOR w. constant
shift left log	sll $1, $2, 10	$1 = $2 << 10	Shift left by constant
shift right log	srl $1, $2, 10	$1 = $2 >> 10	Shift right by constant
shift rt. Arith	sra $1, $2, 10	$1 = $2 >> 10	Shift rt. (sign extend)
shift left var	sllv $1, $2, $3	$1 = $2 << $3	Shift left by variable
shift right var	srlv $1, $2, $3	$1 = $2 >> $3	Shift right by variable
shift rt. arith	srav $1, $2, $3	$1 = $2 >> $3	Shift right arith. var
load upper imm	lui $1, 40	$1 = 40 << 16	Places imm in upper 16b

Loading a 32 bit Constant

- MIPS only has 16 bits of immediate value
- Could load from memory but still have to generate memory address

- Use `lui` and `ori` to load `0xdeadbeef` into $`a0`
  - **`lui $a0, 0xdead            # $a0 = dead0000`**
  - **`ori $a0, $a0, 0xbeef       # $a0 = deadbeef`**