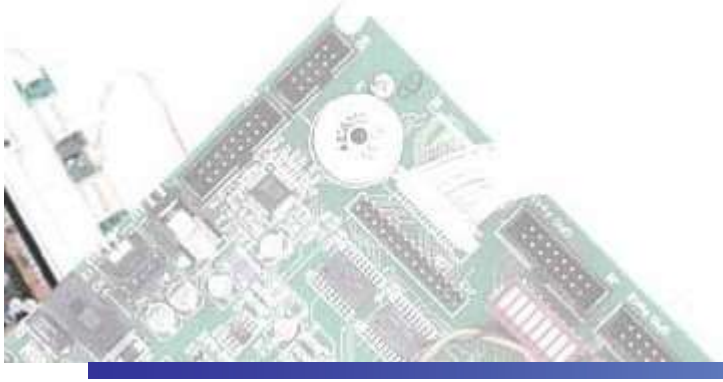
Computer Architecture

Ch. 2 Instructions : Language of the Computer (1)



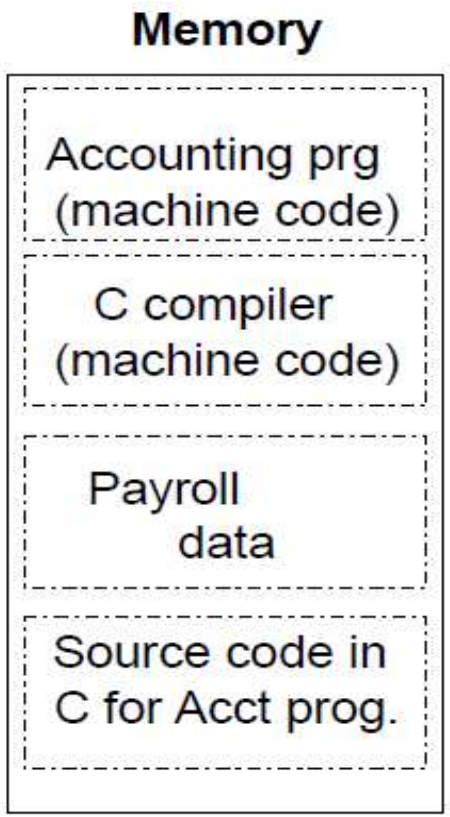
Instructions in Memory (1)

- Instructions are the language of computer
  - vocabulary
  - sentences with a formal grammar
- Same ISA family supports same instructions.
  - Intel x86 family, ARM, MIPS, DEC Alpha, Atmel AVR, ...
  - supports backward compatibility
- Different computers have different instruction sets
  - but with many aspects in common
- Early computers had very simple instruction sets
  - simplified implementation
- Ever since, the ISA have tended to be complicated
  - called CISC (Complex Instruction Set Computer)
- Many modern computers have simple instruction sets
  - called RISC (Reduced Instruction Set Computer)

Instructions in Memory (2)

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data

- Stored-program concept
  - Programs can be shipped as files of binary numbers – binary compatibility
  - Computers can inherit ready-made software provided they are compatible with an existing ISA



A Basic MIPS Instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

```
add   a, b, c      #  a is the sum of b and c
```

Machine code: (hardware-friendly machine instructions)

00000010001100100100000000100000

Translate the following C code into assembly code:

a = b + c + d + e;

Example

C code a = b + c + d + e;

translates into the following assembly code:

```
add  a, b, c          add  a, b, c
add  a, a, d    or    add  f, d, e
add  a, a, e          add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

Subtract Example

C code f = (g + h) – (i + j);

Assembly code translation with only add and sub instructions:

Subtract Example

C code f = (g + h) – (i + j);
translates into the following assembly code:

```
add  t0, g, h              add  f, g, h
add  t1,  i, j      or     sub  f, f, i
sub  f,   t0, t1           sub  f, f, j
```

- Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later

Operands

- In C, each “variable” is a location in memory
- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers
- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed… there can be only so many scratchpad registers

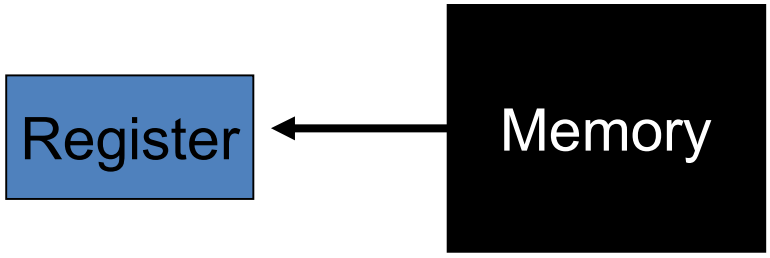
Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word
- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)…

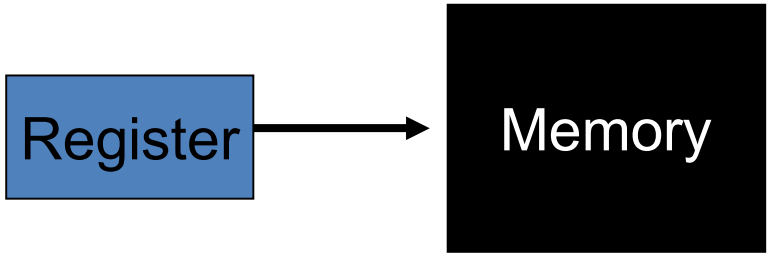
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



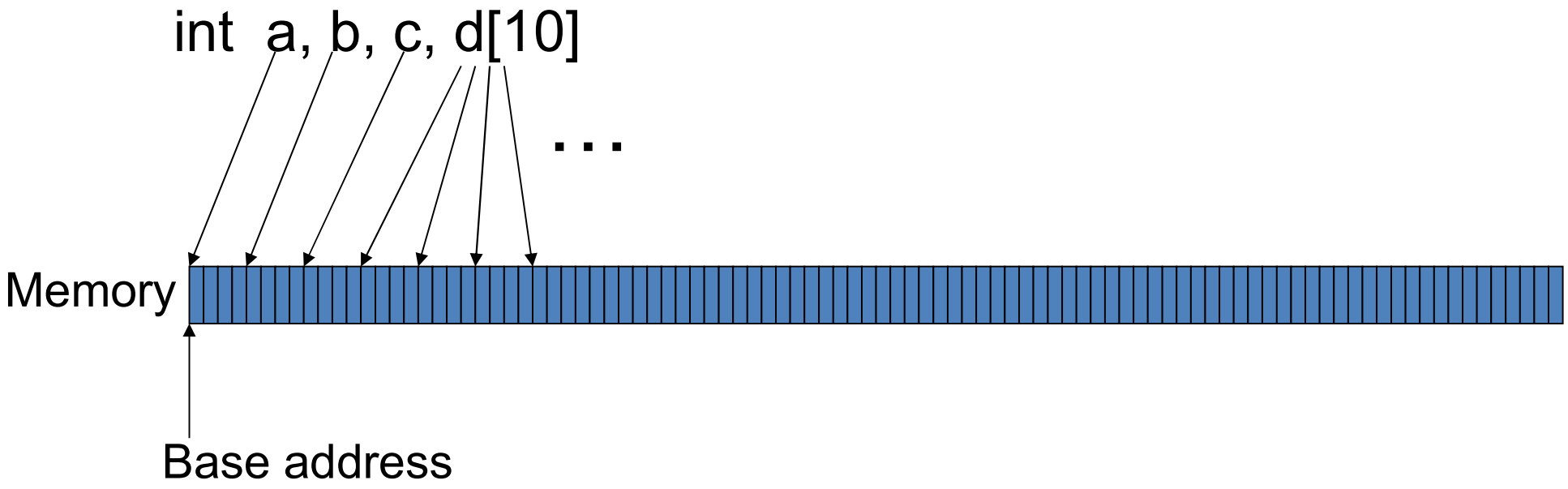
Store word
sw $t0, memory-address



How is memory-address determined?

Memory Address

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions



Immediate Operands

- An instruction may require a constant as input
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

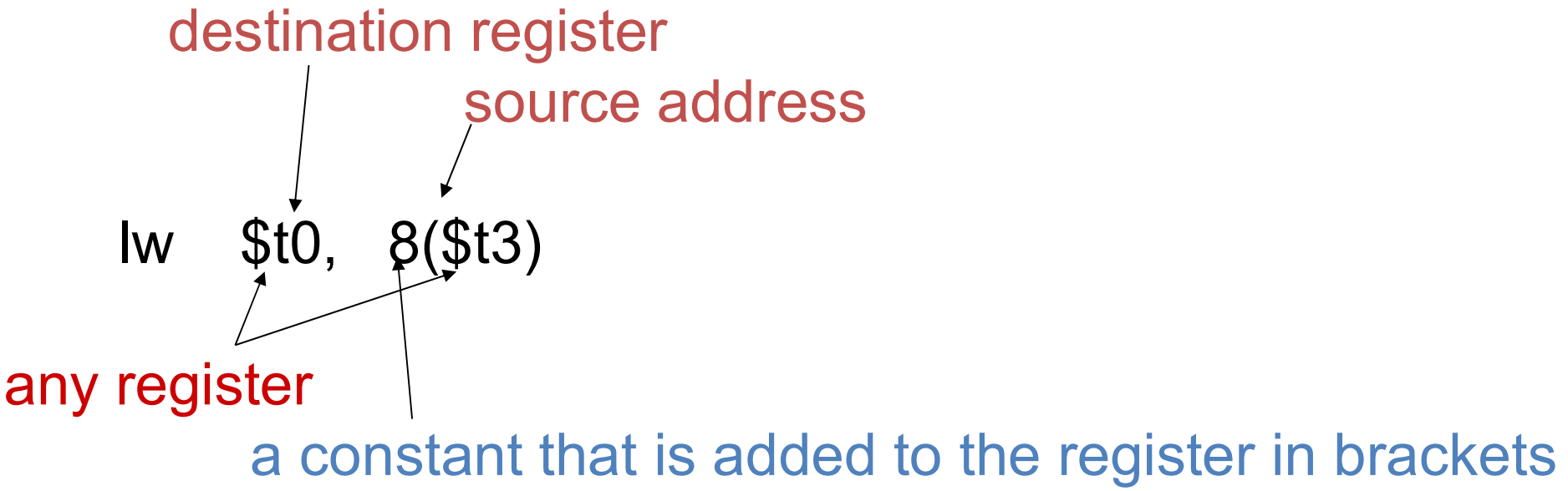
```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

Example

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

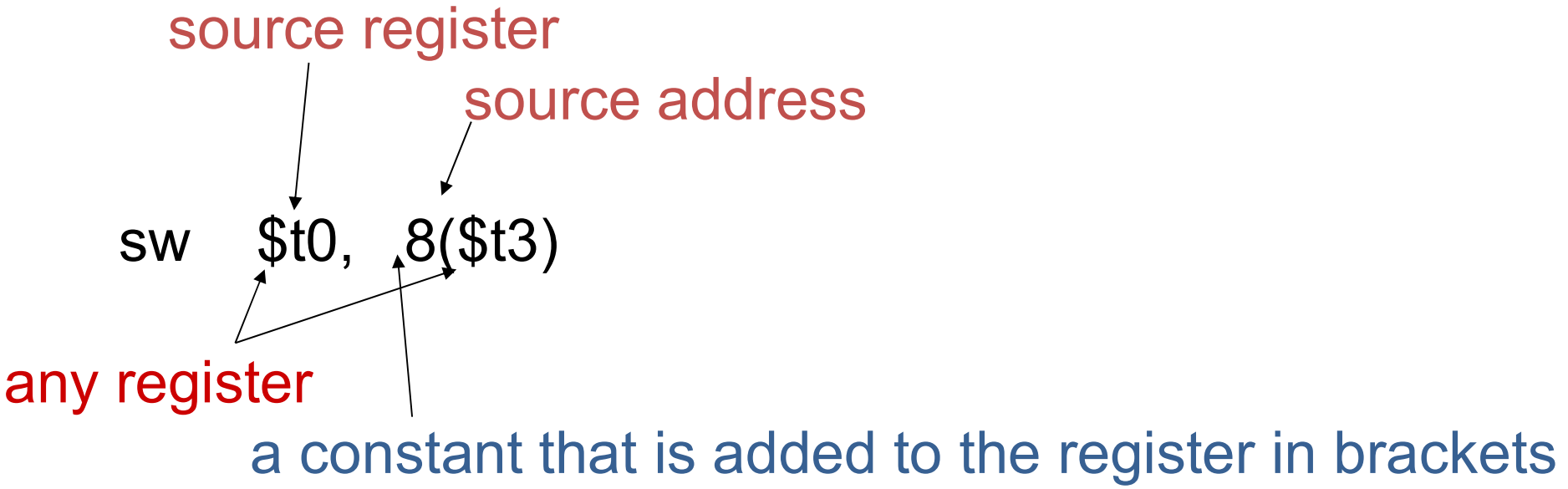
Memory Instruction Format

- The format of a load instruction:



Memory Instruction Format

- The format of a store instruction:



Example

Convert to assembly:

C code: d[3] = d[2] + a;

Example

Convert to assembly:

C code: d[3] = d[2] + a;

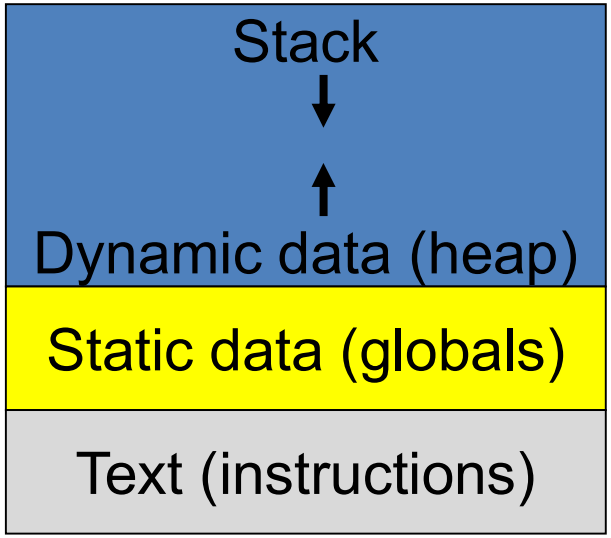
Assembly: # addi instructions as before

```
lw    $t0, 8($s4)    #  d[2] is brought into $t0
lw    $t1, 0($s1)    #   a  is brought into $t1
add   $t0, $t0, $t1   #  the sum is in $t0
sw    $t0, 12($s4)   #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- $gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



Another Version

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly:

```
lw    $t0, 20($gp)    #  d[2] is brought into $t0
lw    $t1, 0($gp)     #   a  is brought into $t1
add   $t0, $t0, $t1   #  the sum is in $t0
sw    $t0, 24($gp)    #  $t0 is stored into d[3]
```

Recap – Numeric Representations

- Decimal $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)

  $0x\ 23$ or $23_{hex} = 2 \times 16^1 + 3 \times 16^0$

  0-15 (decimal) → 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast (and the rest right)
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

MIPS Instruction Set Architecture

- One of the Pioneering RISC Instruction Set Architectures
  - Small repertoire of instructions in a uniform format
  - Pipelined execution
  - Cache memory
  - Load/store architecture
- Starts with a 32-bit architecture, later extended to 64-bit
- Even currently used in many embedded applications
  - Game consoles – Nintendo 64, PlayStation, PlayStation 2, etc
  - Network devices – IP phone, WLAN Access points, etc
  - Residential Devices – High Definition TV, Digital Photo Frame,
  - etc
- See
  - MIPS Reference Data Card and Appendixes B and E
  - *www.mips.com*

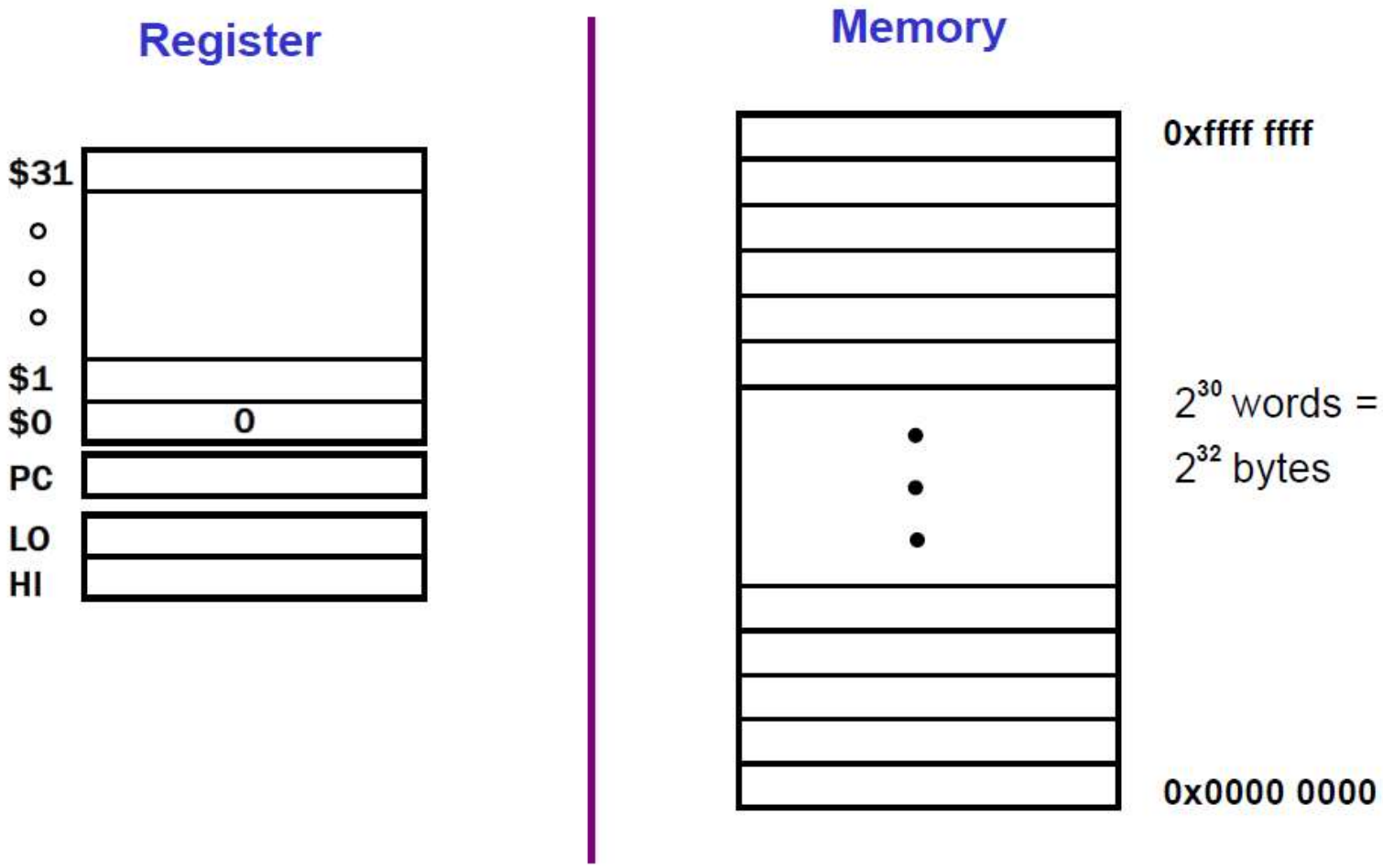
MIPS ISA – Very Regular Formats

- Instruction Categories
  - Arithmetic/Logic instructions
  - Data Transfer (Load/Store) instructions
  - Conditional branch instructions
  - Unconditional jump instructions

- Only 3 instruction formats

Name	Fields						Comments
Field size	**6bits**	**5bits**	**5bits**	**5bits**	**5bits**	**6bits**	**All MIPS insturctions 32 bits**
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

MIPS ISA State (Register & Memory)



MIPS Arithmetic Instructions (1)

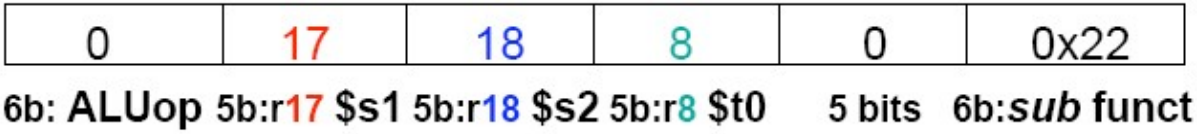
- MIPS assembly language **arithmetic** statement

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

- Each **arithmetic** instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)

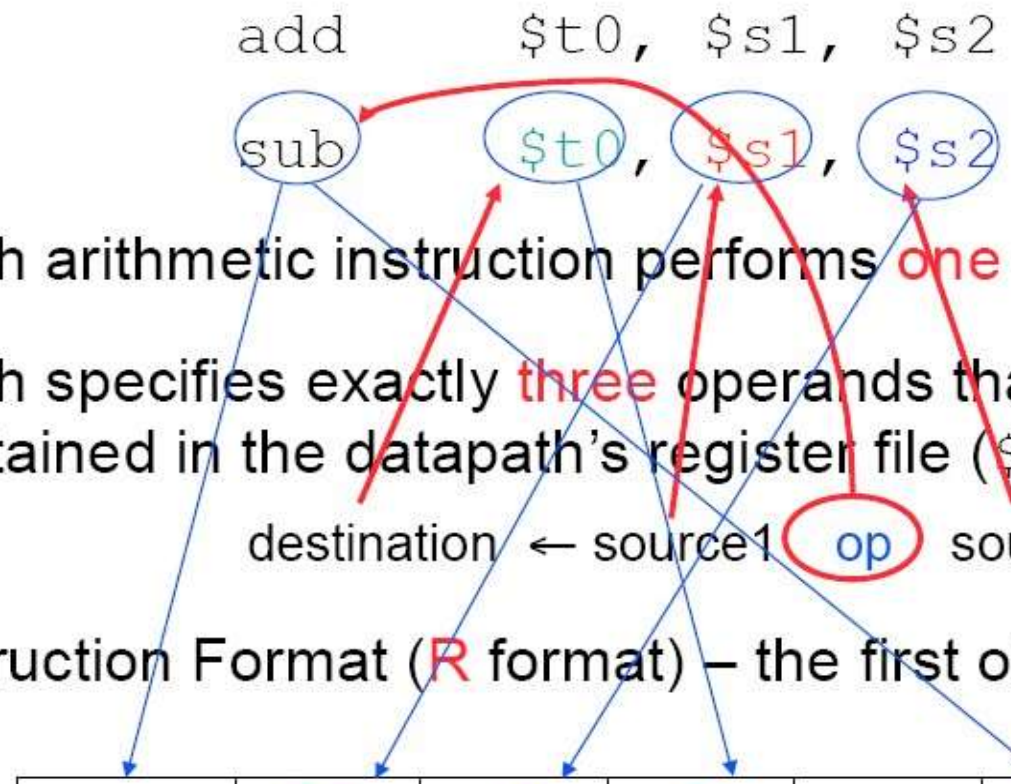
destination ← source1 op source2

- Instruction Format (R format) – the first of only 3 formats
  { R for Register – used by roughly half of all executed instructions }

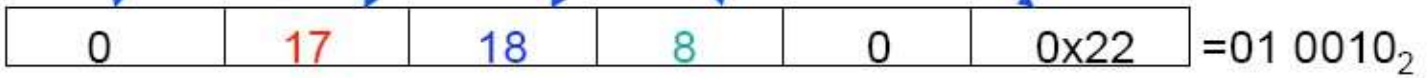


MIPS Arithmetic Instructions (2)

- MIPS assembly language arithmetic statement



- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)
- Instruction Format (R format) – the first of only 3 formats

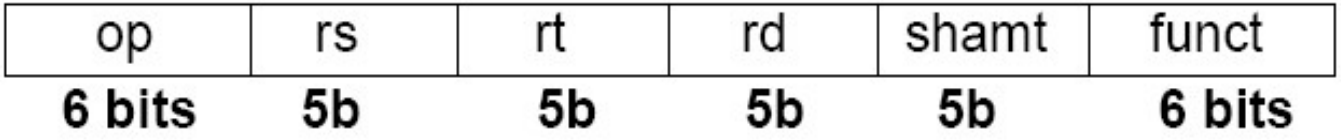


6b: ALUop 5b:r17 $s1 5b:r18 $s2 5b:r8 $t0 5 bits 6b:*sub* funct

(Here, 6b means field in instruction Is 6 bits long, $s = saved register & $t = temporary value in unsaved register)

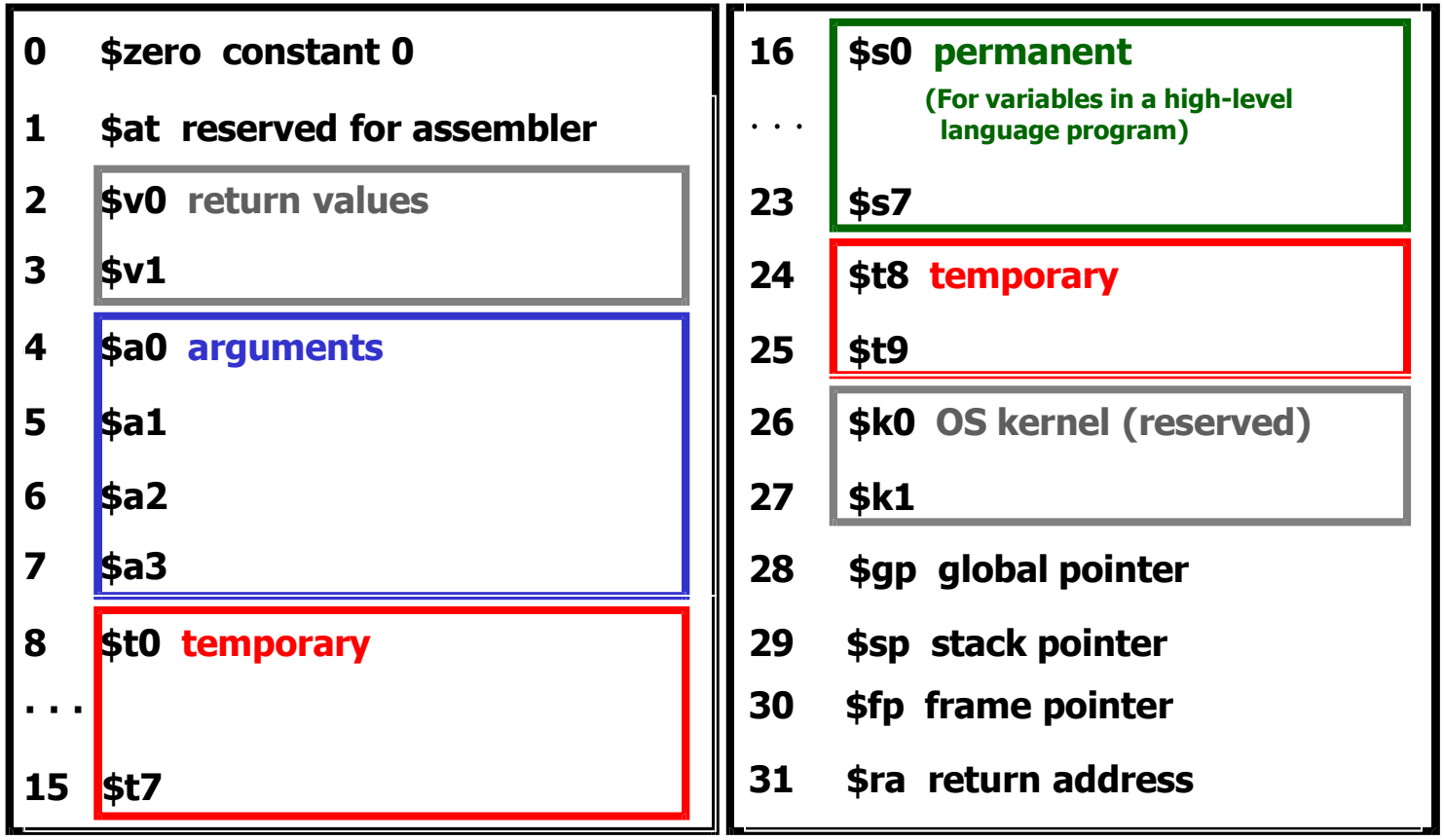
MIPS Instruction Fields: R-format

- MIPS fields are given names to make them easier to refer to



op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS Register Usage (Software Convention for Interoperability)

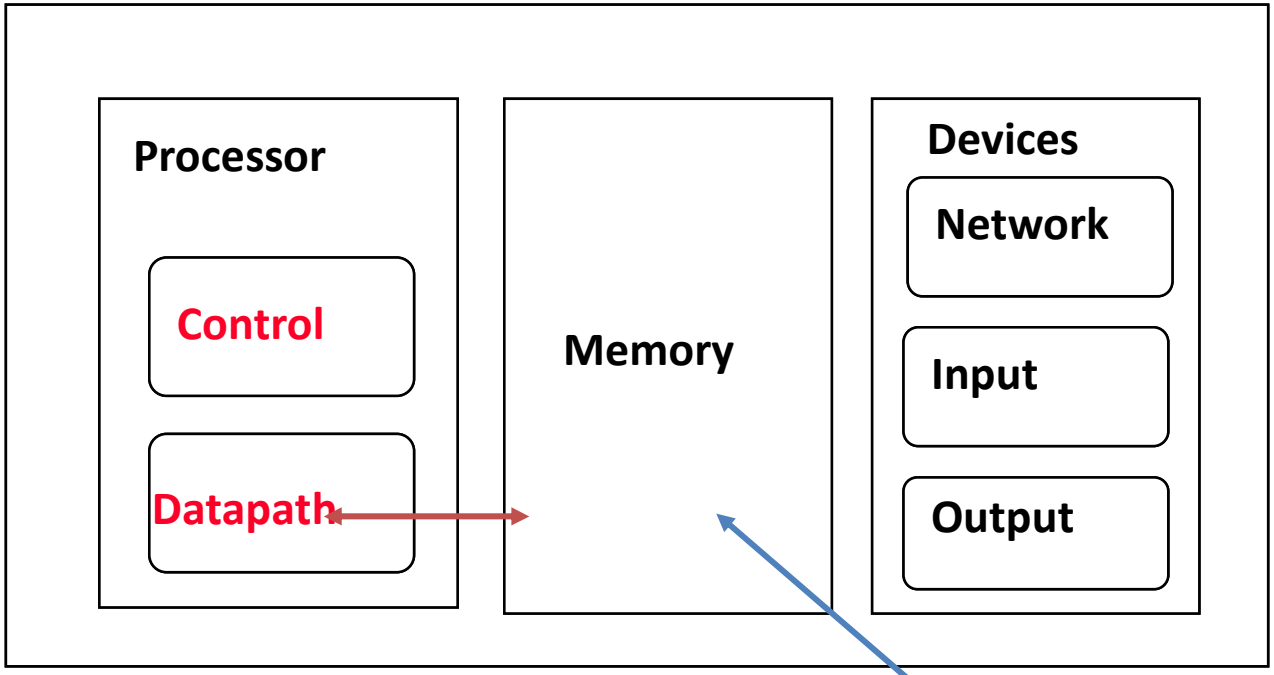


Registers

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

Registers vs. Memory

- Arithmetic instructions *operands* must be in registers
  - only thirty-two registers are provided

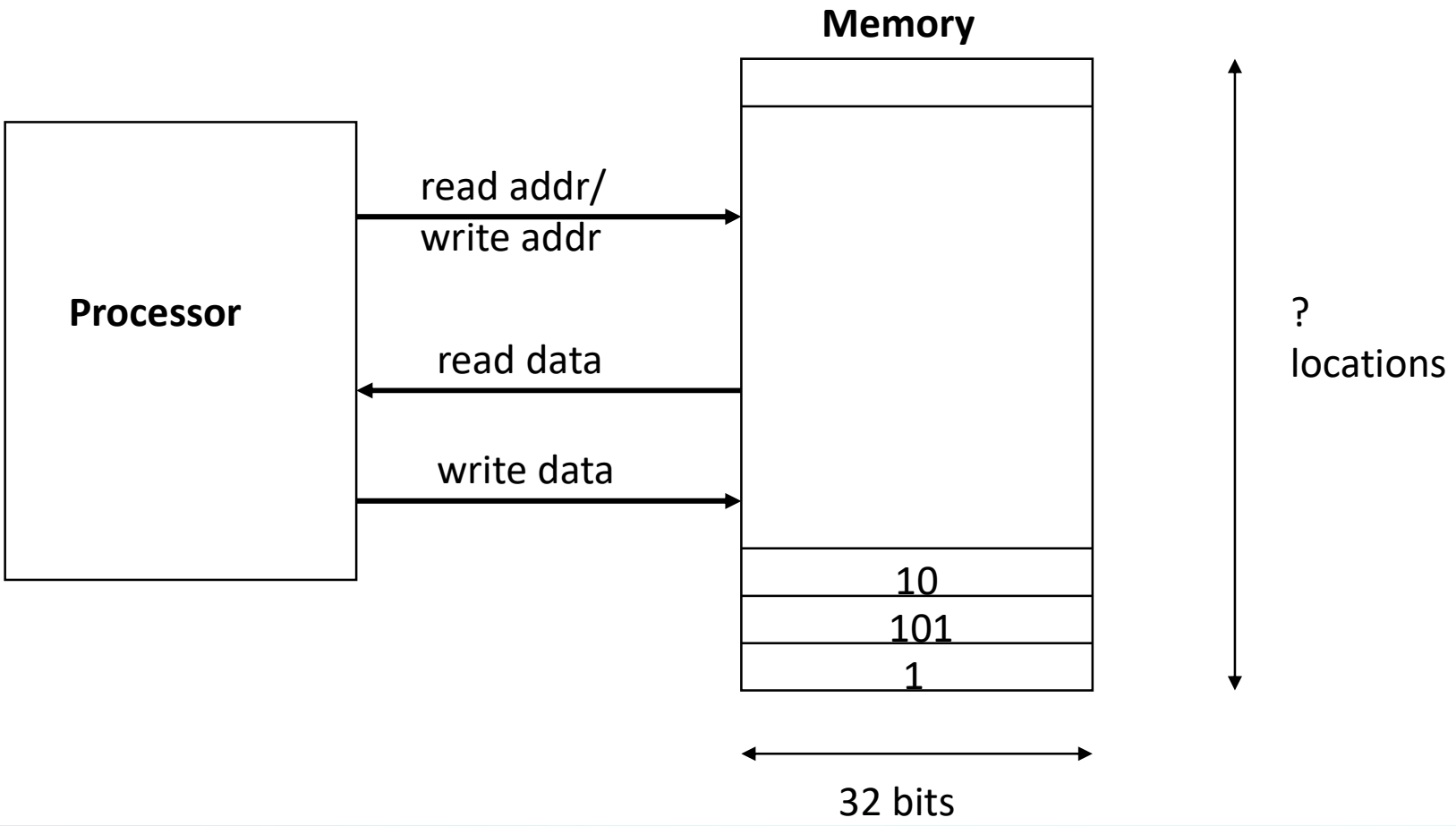


- Compiler associates variables with registers

What about programs with lots of variables?

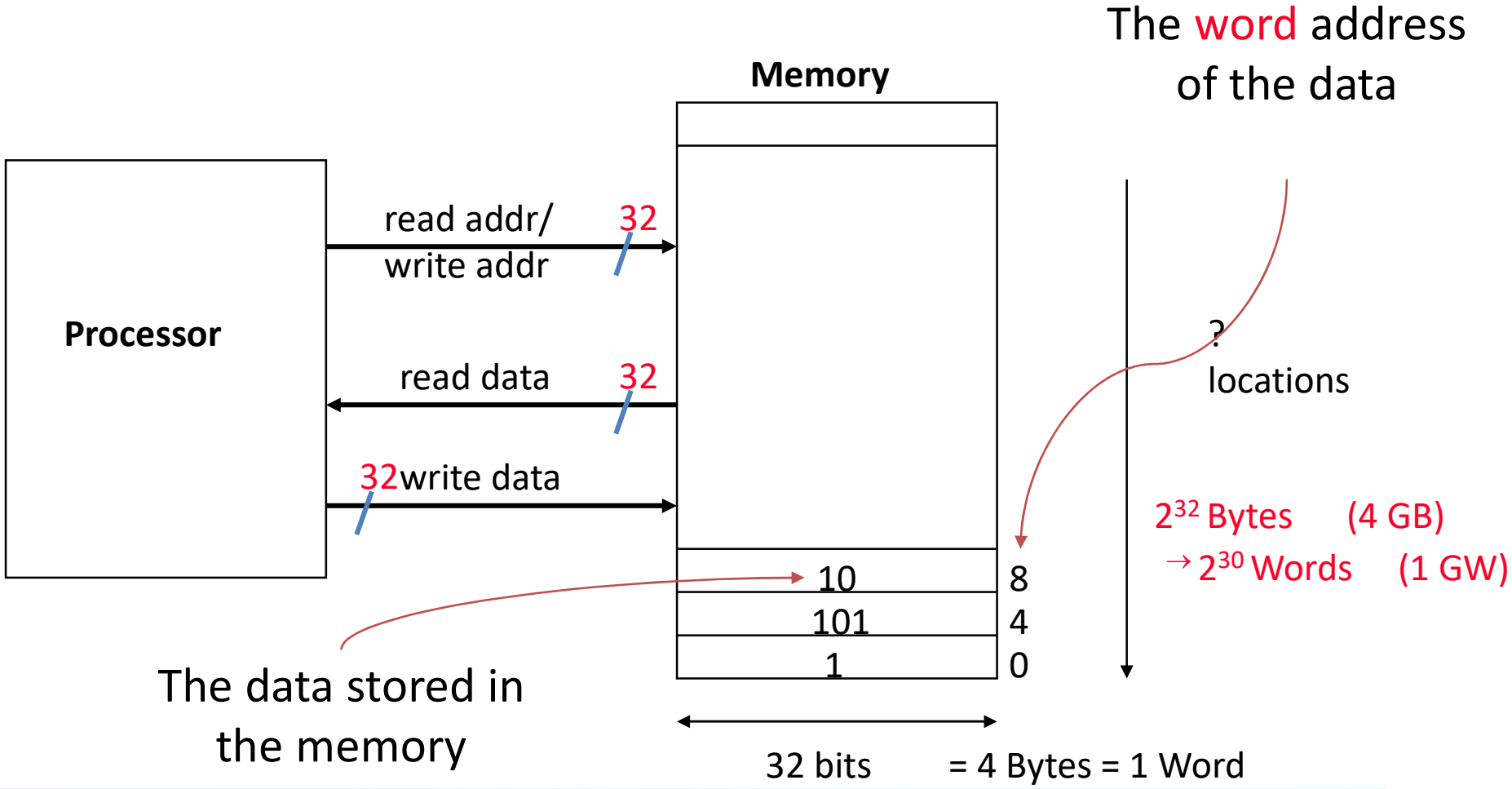
Processor – Memory Interconnections

- Memory is a large, single-dimensional array
- An address acts as the index into the memory array



Processor – Memory Interconnections

- Memory is a large, single-dimensional array
- An address acts as the index into the memory array



MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

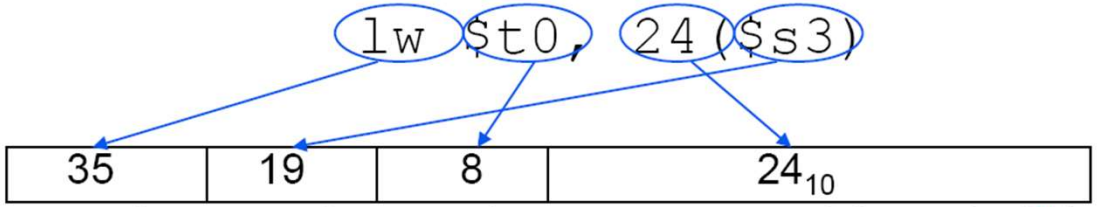
```
lw  $t0, 4($s3)   #load word from memory

sw  $t0, 8($s3)   #store word to memory
```

- The data are loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address

- The memory address – a 32-bit address – is formed by adding the contents of the base address register ($s3 in these examples) to the offset value (here 4 and 8 Bytes)
  - A 16-bit field means access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($=\pm 2^{15}$ or 32,768 bytes) of the address in the base register

Load Instruction

- Load/Store Instruction Format (I format) {2nd format of 3}:



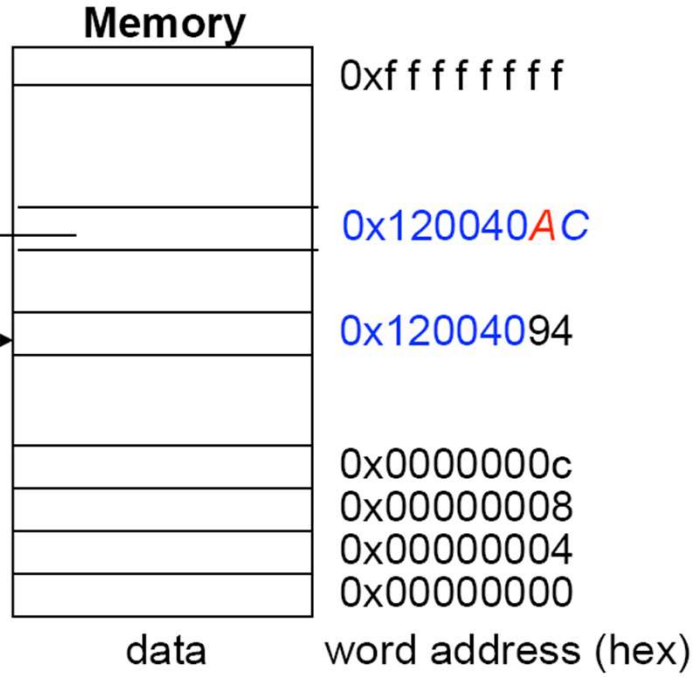
24_{10} + $s3 =

0... 0001 1000 = $24_{10} = 18_{16}$

+ . . . 1001 0100 = 0x120040$94_{16}$ $t0

. . . 1010 1100 = ……AC_{16} $s3

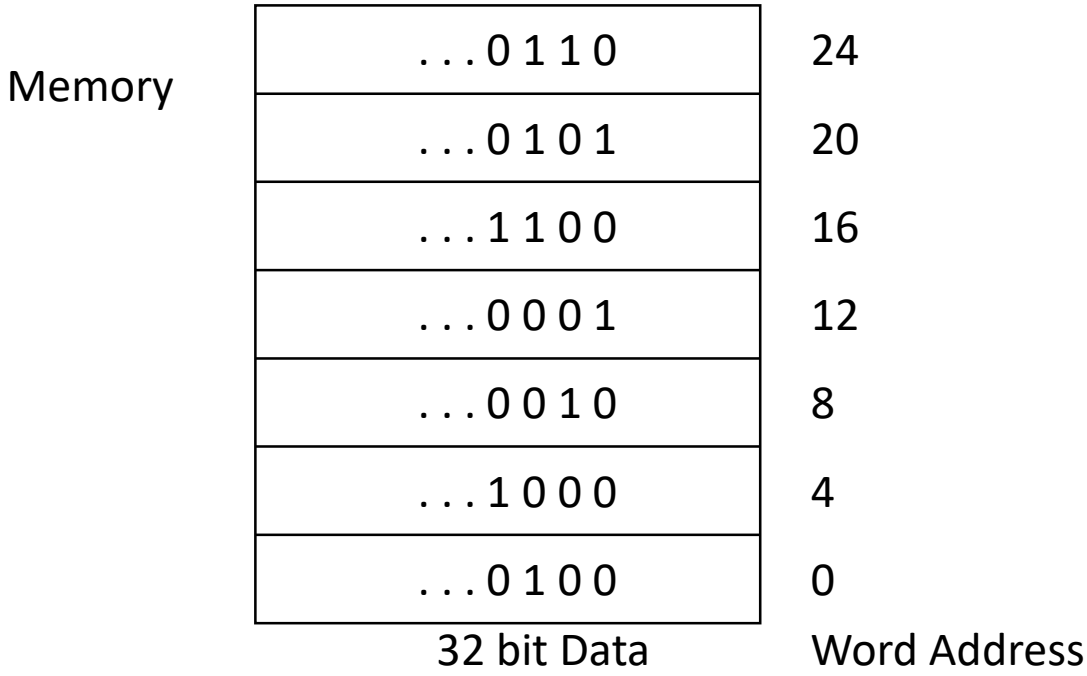
0x120040*AC*



MIPS Memory Addressing

- The memory address is formed by summing the constant portion of the instruction and the contents of the second (base) register

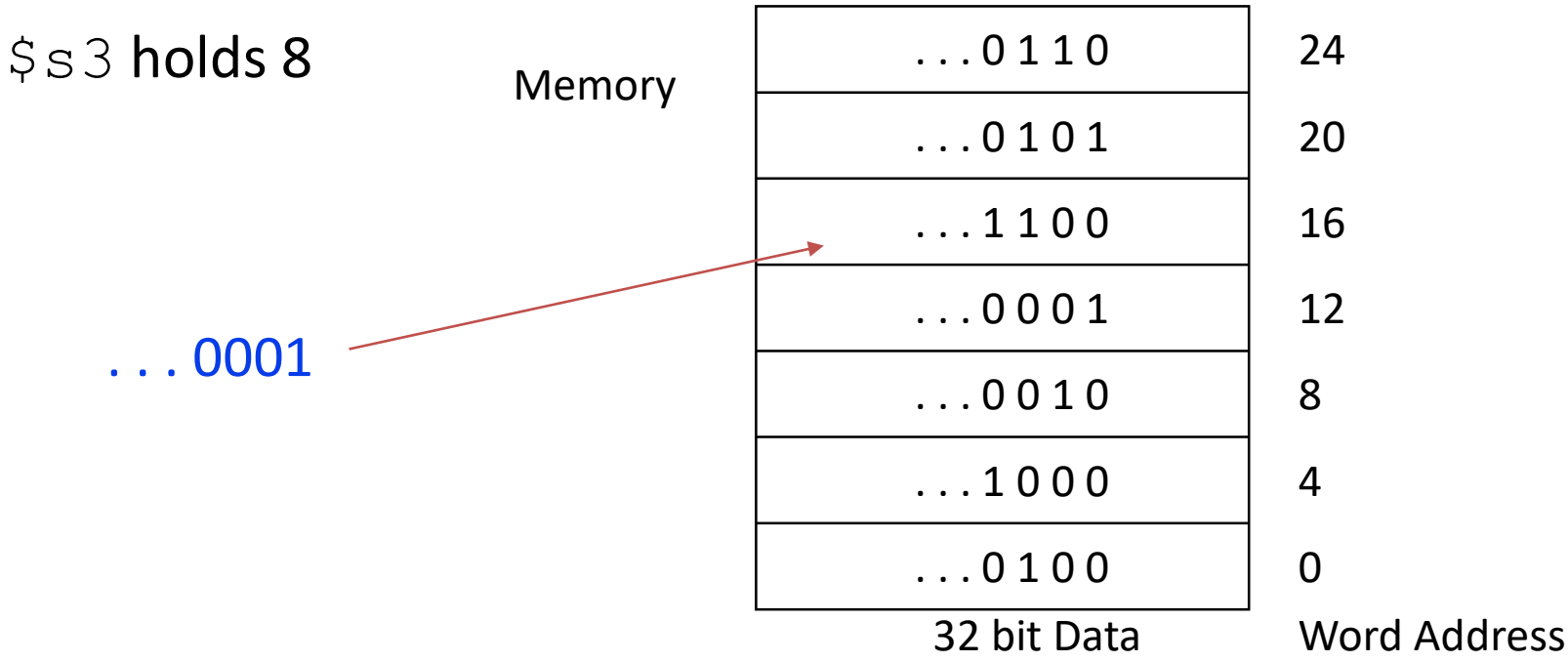
`$s3` holds 8




```
lw    $t0, 4($s3)   #what? is loaded into $t0
sw    $t0, 8($s3)   #$t0 is stored where?
```

MIPS Memory Addressing

- The memory address is formed by summing the constant portion of the instruction and the contents of the second (base) register

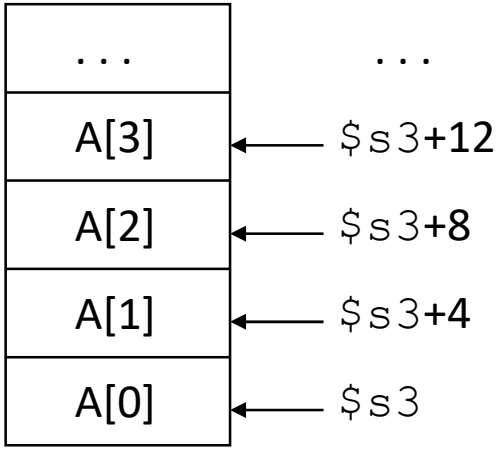



```
lw    $t0, 4($s3)   #what? is loaded into $t0  ...0001
sw    $t0, 8($s3)   #$t0 is stored where?  in location 16
```

Compiling with Loads and Stores

- Assuming variable b is stored in `$s2` and that the base address of array A is in `$s3`, what is the MIPS assembly code for the C statement

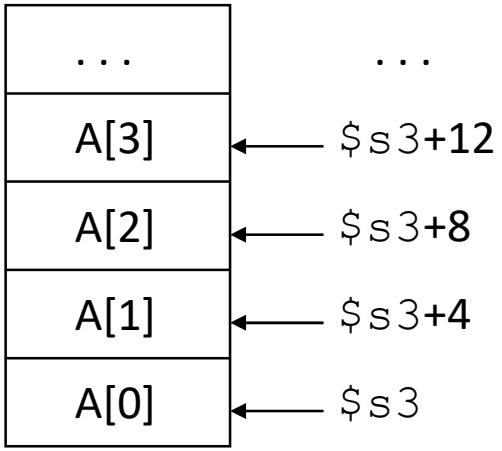
```
A[8] = A[2] - b
```



Compiling with Loads and Stores

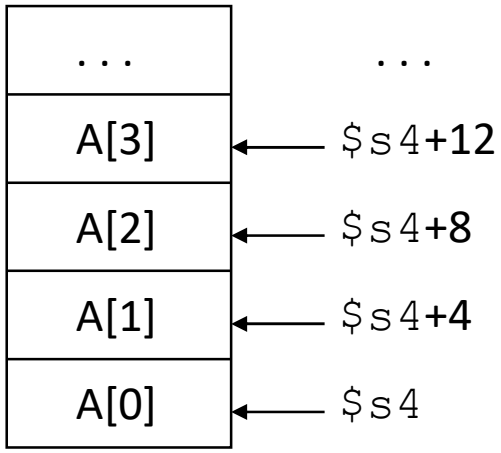
- Assuming variable b is stored in `$s2` and that the base address of array A is in `$s3`, what is the MIPS assembly code for the C statement

```
A[8] = A[2] - b
```




```
lw   $t0, 8($s3)
sub  $t0, $t0, $s2
sw   $t0, 32($s3)
```

Compiling with a Variable Array Index

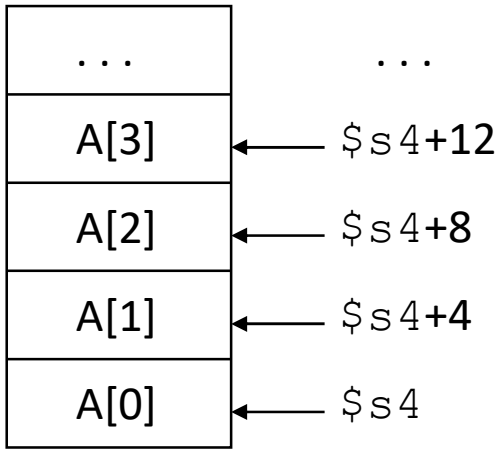


- Assuming that the base address of array A is in register `$s4`, and variables b, c, and i are in `$s1`, `$s2`, and `$s3`, respectively, what is the MIPS assembly code for the C statement

```
c = A[i] - b
```

```
add   $t1, $s3, $s3     #array index i is in $s3
add   $t1, $t1, $t1     #temp reg $t1 holds 4*i
```

Compiling with a Variable Array Index



- Assuming that the base address of array A is in register `$s4`, and variables b, c, and i are in `$s1`, `$s2`, and `$s3`, respectively, what is the MIPS assembly code for the C statement

```
c = A[i] - b
```

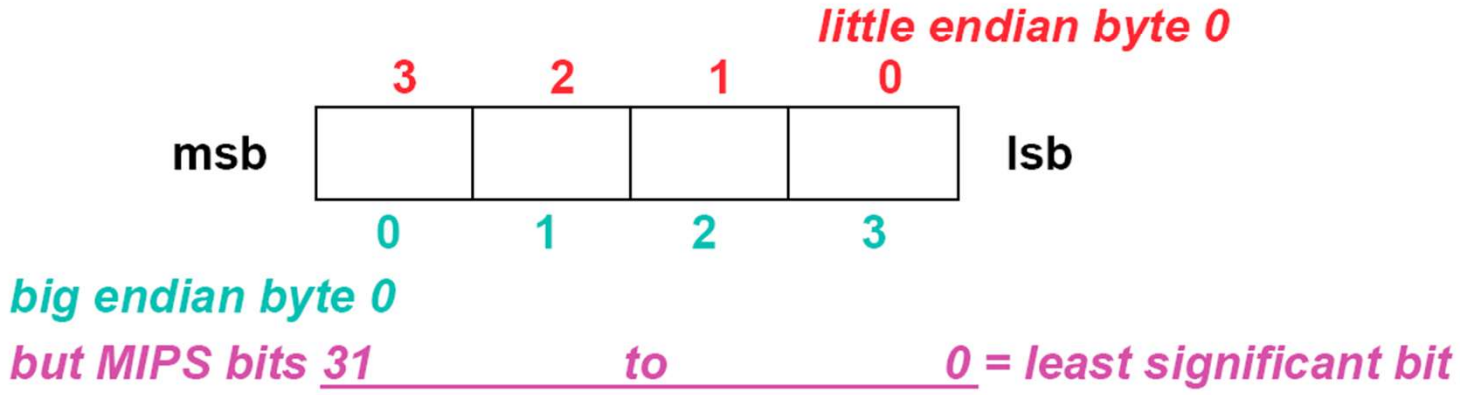
```
add   $t1, $s3, $s3     #array index i is in $s3
add   $t1, $t1, $t1     #temp reg $t1 holds 4*i
add   $t1, $t1, $s4     #addr of A[i] now in $t1
lw    $t0, 0($t1)
sub   $s2, $t0, $s1
```

Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address (text L→R)

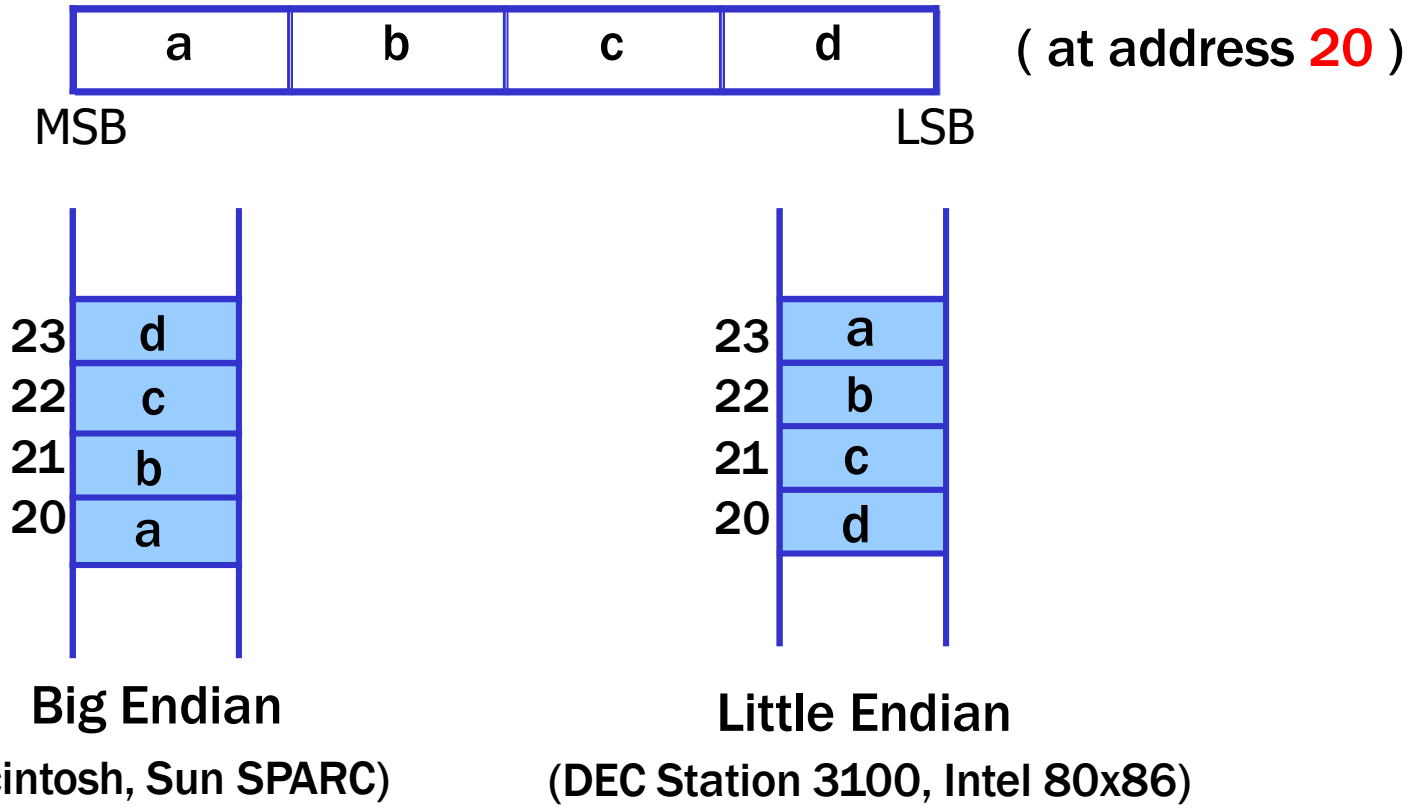
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



More about Endian

- Big Endian vs. Little Endian



Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb $t0,1($s3)   #load byte from memory

sb $t0,6($s3)   #store byte to memory
```

- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

More about Loads and Stores

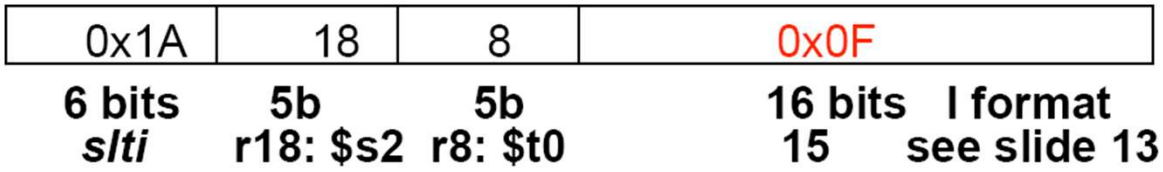
- All memory accesses are exclusively through loads and stores (*load-store* architecture)
- Alignment restriction
  - Word addresses must be multiples of 4
  - Halfword (= 2 bytes) addresses must be multiples of 2
- Partial word (halfword or byte) loads from memory
  - Sign-extended for *signed* operations
  - Zero-extended for *unsigned* operations

MIPS Immediate Instructions (1)

- Small constants are used often in typical code
- Possible approaches for fast access to tiny constants?
  - put "typical constants" in memory and load them NO!
  - create hard-wired registers (like $zero) for constants like 1 NO!
  - have special "immediate" instructions that contain constants !

```
addi  $sp, $sp, 4      #$sp = $sp + 4

slti  $t0, $s2, 15     #$t0 = 1 if $s2<15
```

- Machine format (I format):



- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{15}-1$ to -2^{15}

MIPS Immediate Instructions (2)

- How about larger constants (e.g. 32-bit) into a register?
- *lui* ("load upper immediate") instruction

```
lui $t0, 1010101010101010
```

0x0F	0	8	1010101010101010_2
6 bits	**5b**	**5b**	**16 bits I format**
lui		**r8: $t0**	1010…1010 **see slide 13**

Then must get the lower order bits right, use

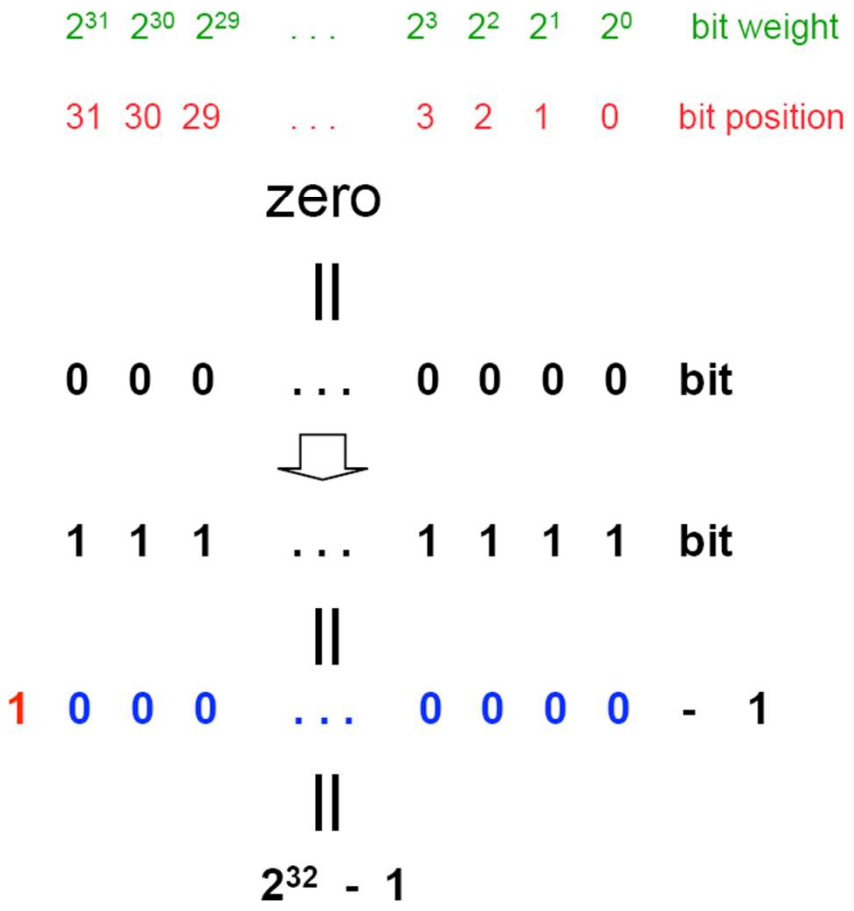
```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

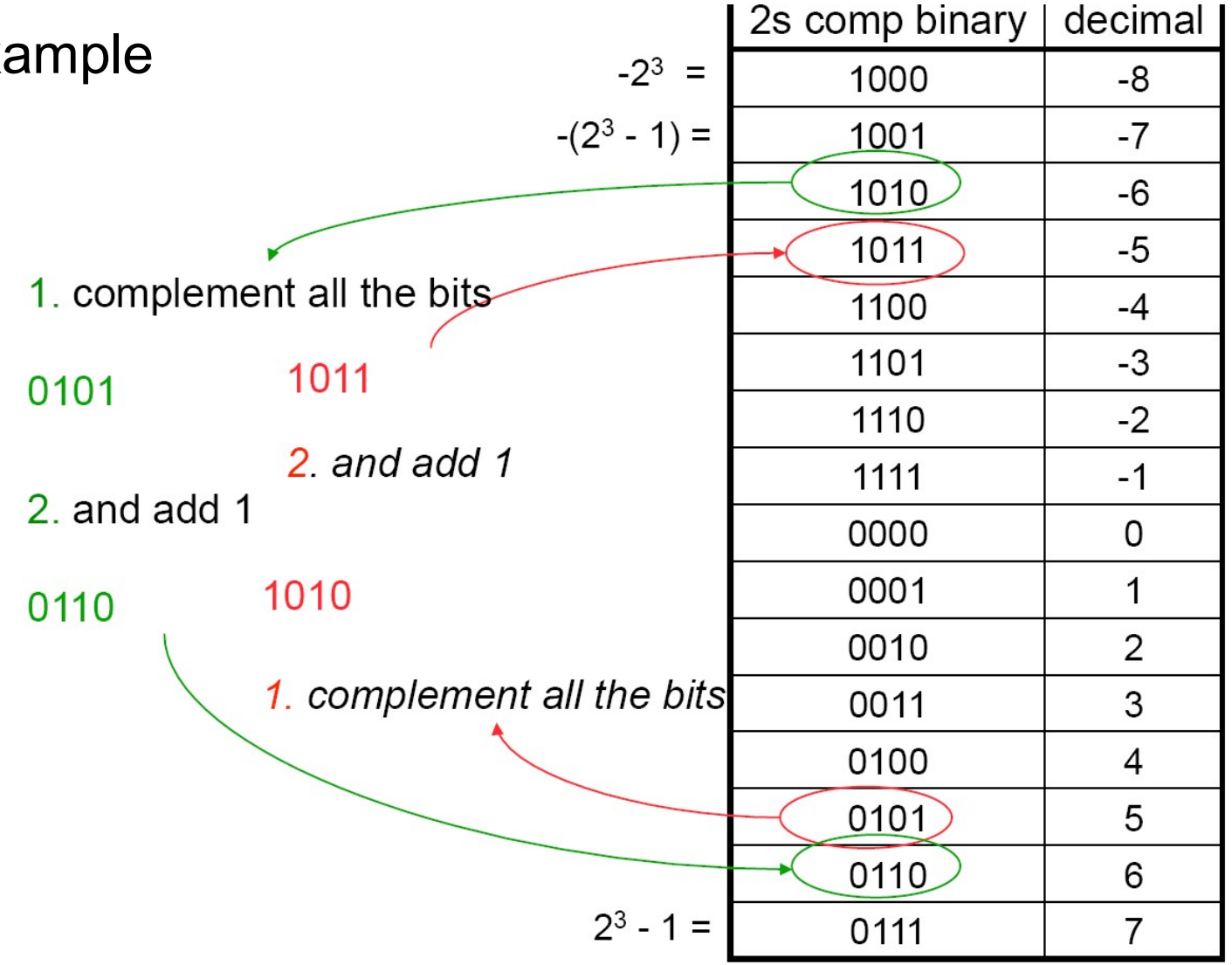
Review: Unsigned Binary Repersentation

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	$2^{32} - 4$
0xFFFFFFFD	1…1101	$2^{32} - 3$
0xFFFFFFFE	1…1110	$2^{32} - 2$
0xFFFFFFFF	1…1111	$2^{32} - 1$



Review: Signed Binary Representation

- 4-bit Example



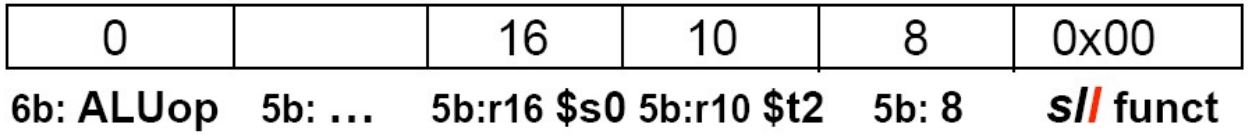
	2s comp binary	decimal
-2^3 =	1000	-8
$-(2^3 - 1)$ =	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1$ =	0111	7

MIPS Shift Operations

- Need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8     #$t2 = $s0 << 8 bits

srl $t2, $s0, 8     #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)



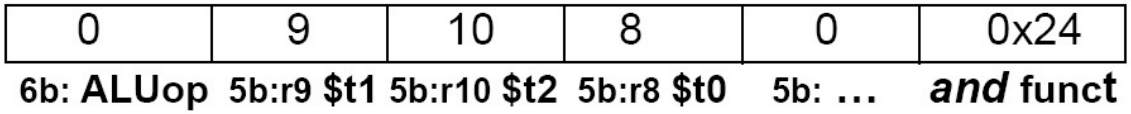
- Such shifts are called logical because they fill with zeros
  - The 5-bit shamt field permits 31 $\{2^5-1\}$ bit shifts of 32-bit values
- Shift right arithmetic (sra) fills with sign-bits, not zeros

MIPS Logical Operations

- There are a number of bit-wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2  #$t0 = $t1 & $t2
or  $t0, $t1, $t2  #$t0 = $t1 | $t2
nor $t0, $t1, $t2  #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)




```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Instruction Format (I format)

NOT operation in MIPS

- NOT is useful to invert (negate) bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS supports only NOR 3-operand instruction
  - (a NOR b) == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

MIPS Control Flow Instructions

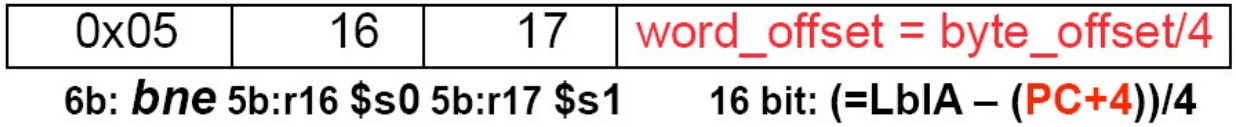
- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl   #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl   #go to Lbl if $s0=$s1
```

  - Ex:

```
        if (i==j) h = i + j;

        bne $s0, $s1, LblA
        add $s3, $s0, $s1
LblA:   ...
```

- Instruction Format (I format):



- How is the branch destination address specified?

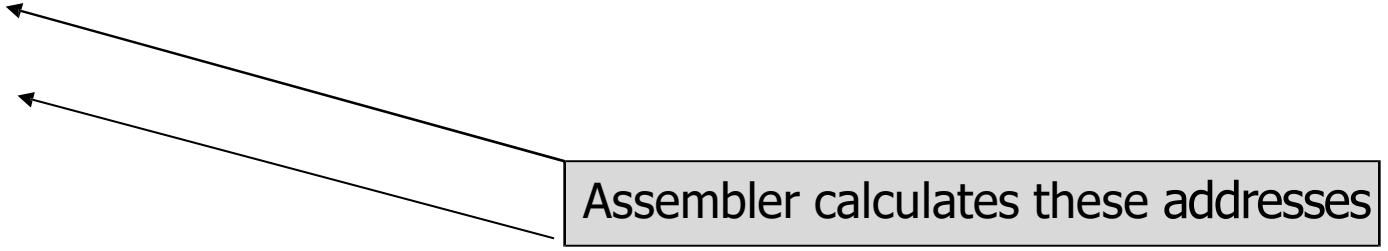
Branch Examples

- Example

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

- Compiled MIPS code

```
    bne $s3,$s4,Else      # go to Else if i≠j
    add $s0,$s1,$s2       # f=g+h (skipped if i≠j)
    j Exit                # go to Exit
Else: sub $s0,$s1,$s2     # f=g-h (skipped if i==j)
Exit:
```

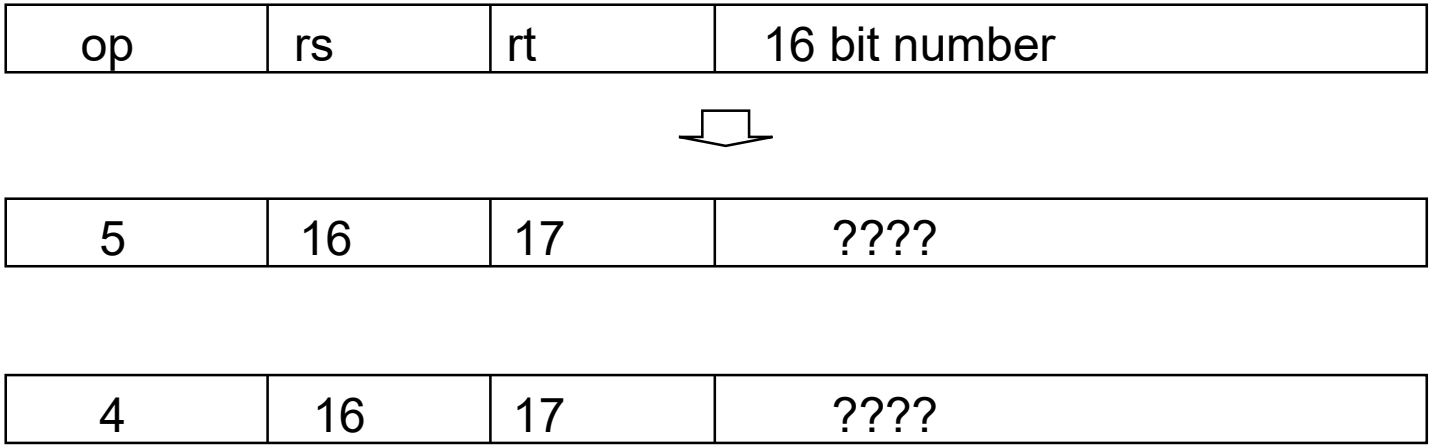


Assembling Branches

- Instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

- Machine Formats:

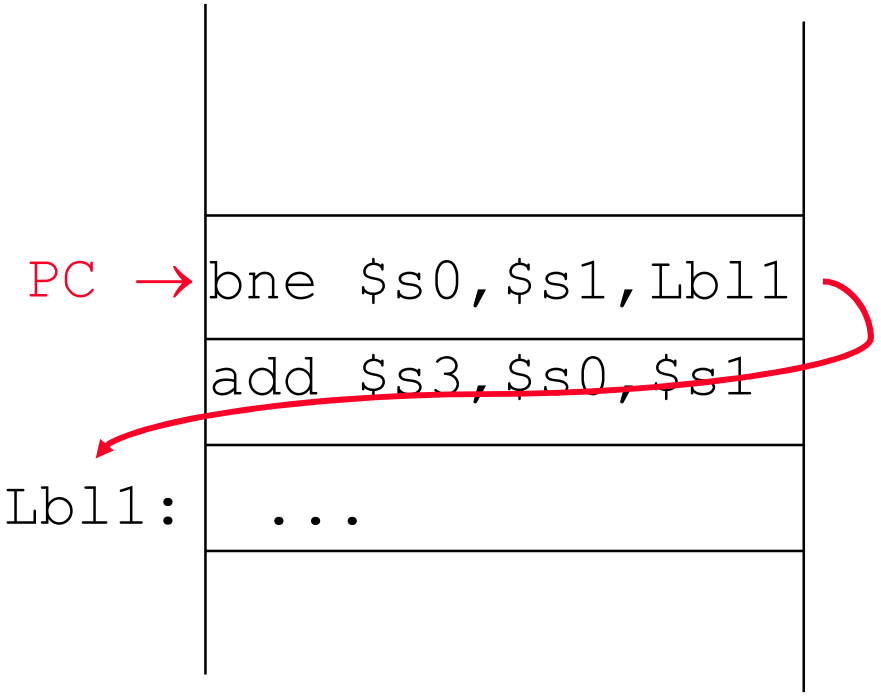


I format

- How is the branch destination address specified?

Specifying Branch Destinations

- Could specify the memory address - but that would require a 32 bit field



- Could use a "base" register and add to it the 16-bit offset

  which register?

  - Instruction Address Register

    (PC = program counter) - its use is automatically implied by branch

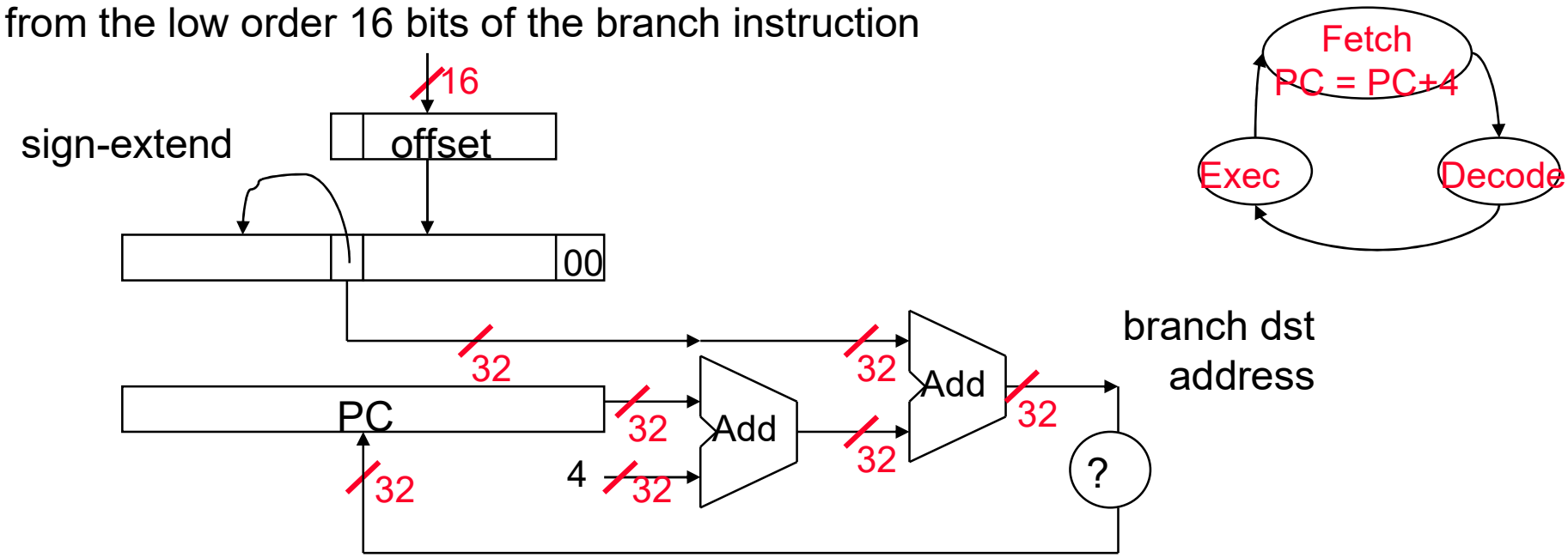
  - PC gets updated (PC+4) during the Fetch cycle so that it holds the address of the next instruction

  limits the branch distance to -2^{15} to $+2^{15}-1$ instr's from the (instruction after the) branch

  - but most branches are local anyway

Disassembling Branch Destinations

- The contents of the updated PC (PC+4) is added to the 16 bit branch offset which is converted into a 32 bit value by
  - concatenating two low-order zeros to make it a word address and then sign-extending those 18 bits
- The result is written into the PC if the branch condition is true - before the next Fetch cycle



Offset Tradeoffs

- Why not just store the word offset in the low order 16 bits? Then the two low order zeros wouldn't have to be concatenated, it would be less confusing, …

- That would limit the branch distance to -2^{13} to $+2^{13}-1$ instructions from the (instruction after the) branch
- And concatenating the two zero bits costs us very little in additional hardware and has no impact on the clock cycle time

'*less-than*' Conditional Branch

- We have `bez`, `bnz`, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt (set if less than)`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1 then
                      #  $t0 = 1     else
                      #  $t0 = 0
```

- Instruction format (R format):

- Alternate versions of `slt`

```
slti  $t0, $s0, 25    # if $s0 < 25  then $t0=1 ...
sltu  $t0, $s0, $s1   # if $s0 < $s1 then $t0=1 ...
```

Misc. Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value 0 in register `$zero` to create 4 conditional branch pseudo-operations:
  - less than `blt $s1, $s2, Label`

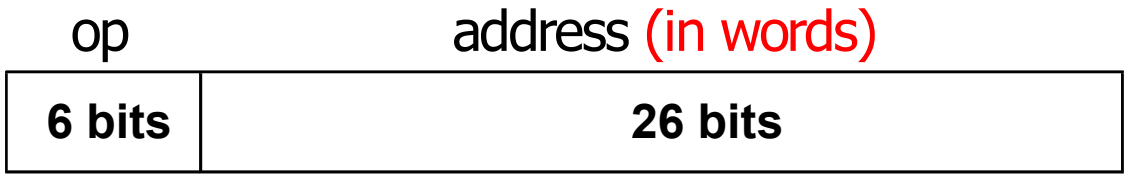
```
slt  $at, $s1, $s2      #$at set to 1 iff
bne  $at, $zero, Label    #$s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions "pseudo-ops" - recognized (and expanded into **`real operations`**) by the assembler
  - It is why the assembler needs a reserved temp register (**`$at`**)

Unconditional Jumps (1)

- J-type instruction format
  - Jump (j and jal) targets could be anywhere in text (code) segment
  - Encode full address in instruction (명령어 주소는 항상 워드 단위로 표현)



cf. jr (jump register) is NOT J-type instruction.

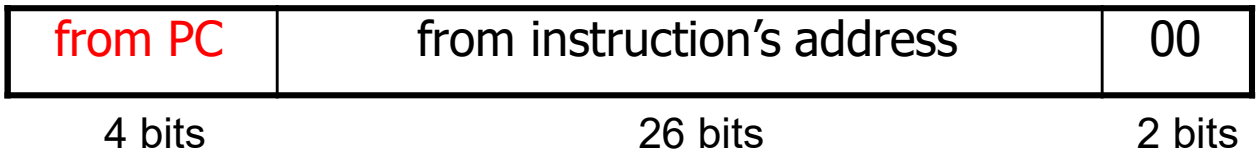
- Direct addressing mode
  - Addressing in jump instruction

```
j   10000     # branch address =10000 (in words)
```

    - Word address
      - Branch address = 10000 * 4
      - `j   10000     # real branch address = 40000 (in bytes)`

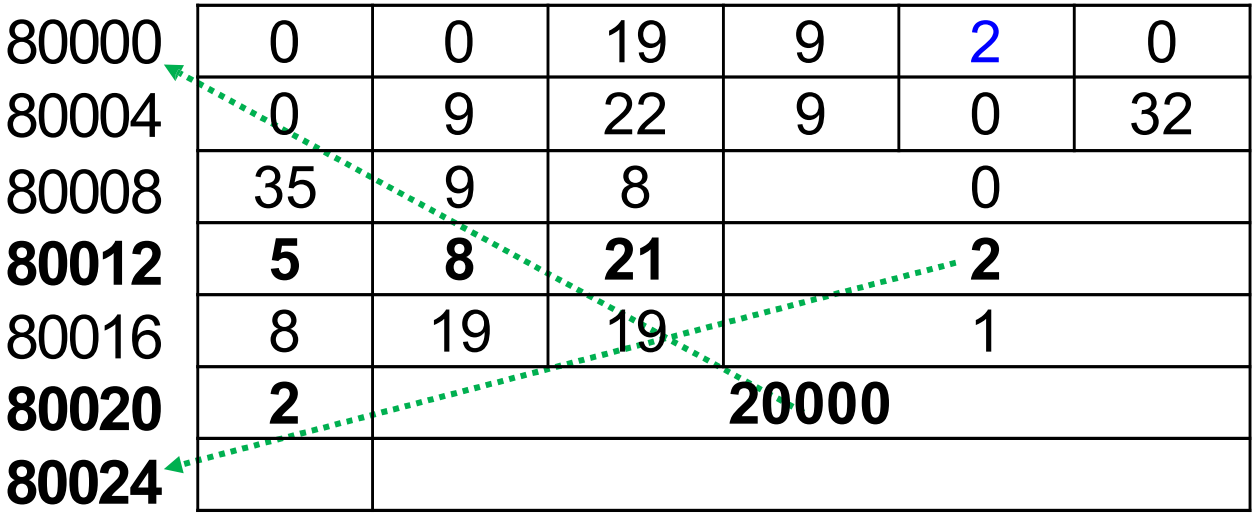
Unconditional Jumps (2)

- MIPS supports Pseudo-direct addressing
  - Upper 4 bits of current PC are used unchanged.
    - Target address = $PC_{31...28}$: (address × 4)
    - Address boundary of 256 MB
  - Target address



Example: Target Addressing with Branch Offset

```
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            # go to Loop
Exit:
```



How to branch far away (over 16-bit offset)

- What if the conditional branch destination is further away than can be captured in 16 bits?
  - cf. Today memory address space is based on 32-bit.

- Assembler calculates branch distance
  - it inserts an unconditional jump to the branch target and inverts the condition

```
            beq  $s0, $s1, L1_too_far_away
```

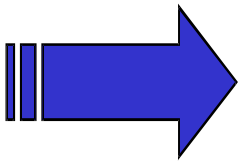
becomes

```
            bne  $s0, $s1, L2
            j    L1_too_far_away
      L2:
```

C vs. Assembly

f = (g + h) – (i + j);

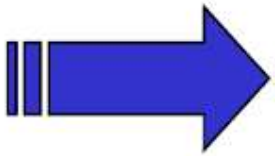
f is mapped to s0
g is mapped to s1
h is mapped to s2
i is mapped to s3
j is mapped to s4



```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

C vs. Assembly

```
g = h + A[i] ;
```

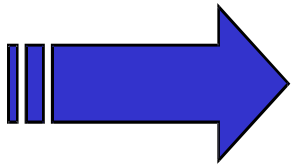


```
add $t1, $s4, $s4
add $t1, $t1, $t1
add $t1, $t1, $s3
lw   $t0, 0($t1)
add $s1, $s2, $t0
```

g is mapped to s1
h is mapped to s2
s3 contains the base address of array A[].
i is mapped to s4.

C vs. Assembly

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

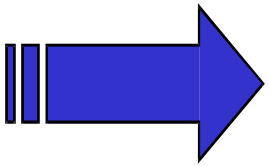


```
    bne $s3, $s4, Else
    add $s0, $s1, $s2
    j   Exit
Else: sub $s0, $s1, $s2
Exit:
```

f is mapped to s0
g is mapped to s1
h is mapped to s2
i is mapped to s3
j is mapped to s4

C vs. Assembly

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



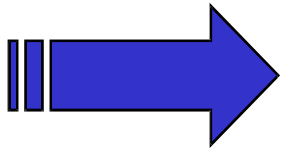
```
Loop:
    add $t1, $s3, $s3
    add $t1, $t1, $t1
    add $t1, $t1, $s6
    lw  $t0, 0($t1)
    bne $t0, $s5, Exit
    add $s3, $s3, $s4
Exit:
    j   Loop
```

i is mapped to s3
j is mapped to s4
k is mapped to s5
s6 contains the base address of array save[].

C vs. Assembly

```
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;
}
```

f is mapped to s0
g is mapped to s1
h is mapped to s2
I is mapped to s3
j is mapped to s4
k is mapped to s5
t2 contains 4



```
    slt   $t3, $s5, $zero
    bne $t3, $zero, Exit
    slt   $t3, $s5, $t2
    beq  $t3, $zero, Exit
    add  $t1, $s5, $s5
    add  $t1, $t1, $t1
    add  $t1, $t1, $t4
    lw   $t0, 0($t1)
    jr    $t0
L0: add  $s0, $s3, $s4
    j Exit
L1: add  $s0, $s1, $s2
    j Exit
L2: sub  $s0, $s1, $s2
    j Exit
L3: sub  $s0, $s3, $s4
Exit:
```