5DV118
Computer Organization and Architecture
Umeå University
Department of Computing Science

Stephen J. Hegner

Topic 2: Instructions

Part A: Basic Concepts

These slides are mostly taken verbatim, or with minor changes, from those prepared by

Mary Jane Irwin (<[:isPlaceholder:]>)

of The Pennsylvania State University

[Adapted from *Computer Organization and Design, 4th Edition*,

Patterson & Hennessy, © 2008, MK]

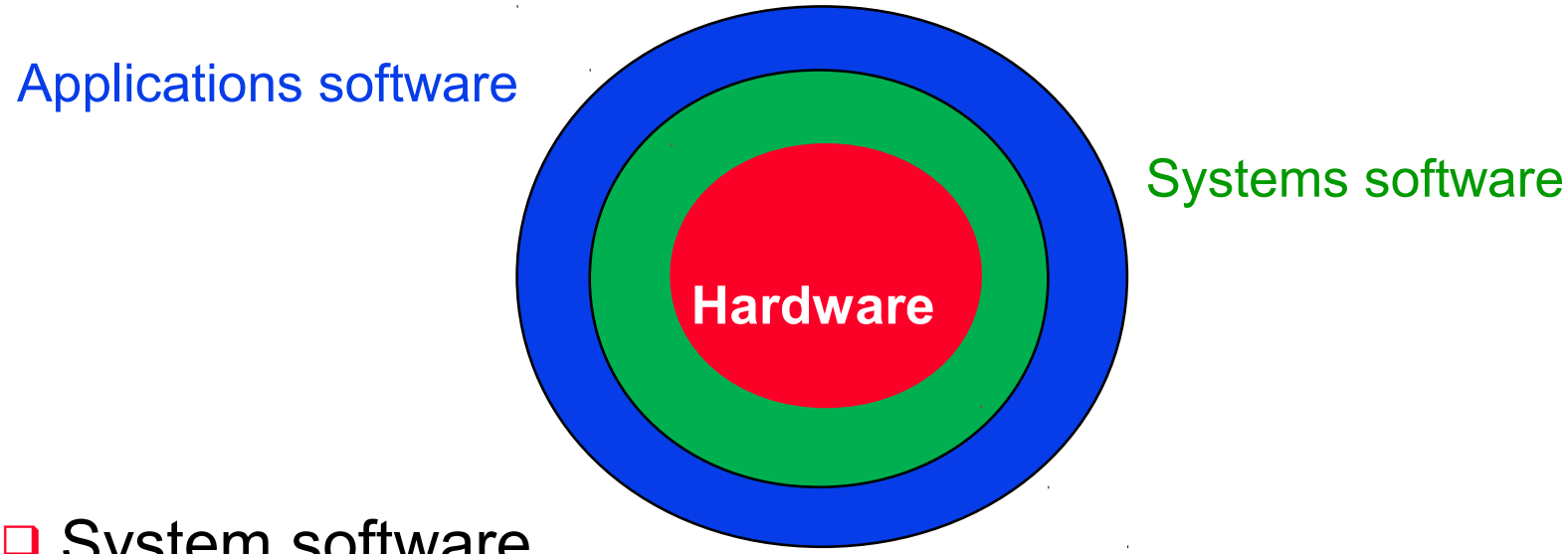
Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - Irwin CSE331 PSU = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at Pennsylvania State University.
  - Irwin CSE431 PSU = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at Pennsylvania State University.
  - Hegner UU = slide by Stephen J. Hegner at Umeå University.

Computer Organization and Design

- This course is all about how computers work
- But what do we mean by a computer?
  - Different types: embedded, laptop, desktop, server, supercomputer
  - Different uses: robotics, graphics, finance, genomics,…
  - Different manufacturers: Intel, IBM, AMD, ARM, Freescale, Fujitsu, TI, Sun (Oracle) , MIPS, NEC, …
  - Different underlying technologies and different costs !
- Best way to learn:
  - Focus on a *specific* instance and learn how it works
  - While learning general principles and historical perspectives

Below the Program



- System software
  - Operating system – supervising program that interfaces the user's program with the hardware (e.g., Linux, MacOS, Windows)
    - Handles basic input and output operations
    - Allocates storage and memory
    - Provides for protected sharing among multiple applications
  - Compiler – translate programs written in a high-level language (e.g., C, Java) into instructions that the hardware can execute

Advantages of High-Level Languages ?

- Higher-level languages

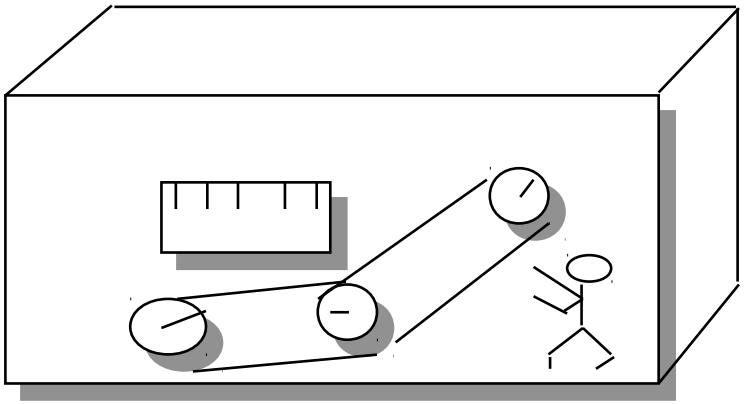
- As a result, very little programming is done today at the assembler level

Advantages of High-Level Languages ?

- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, Java for web programming, …)
  - Improve programmer productivity – more understandable code that is easier to debug and validate
  - Improve program maintainability
  - Allow programs to be independent of the computer on which they are developed (compilers and assemblers can translate high-level language programs to the binary instructions of any machine)
  - Emergence of optimizing compilers that produce very efficient assembly code optimized for the target machine
- As a result, very little programming is done today at the assembler level

Machine Organization

- Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, arithmetic-logic unit (ALU), multiplexors, memories, ...
- The ways those FUs are interconnected
  - e.g., buses
- Logic and means by which information flow between FUs is controlled



- The machine's Instruction Set Architecture (ISA)

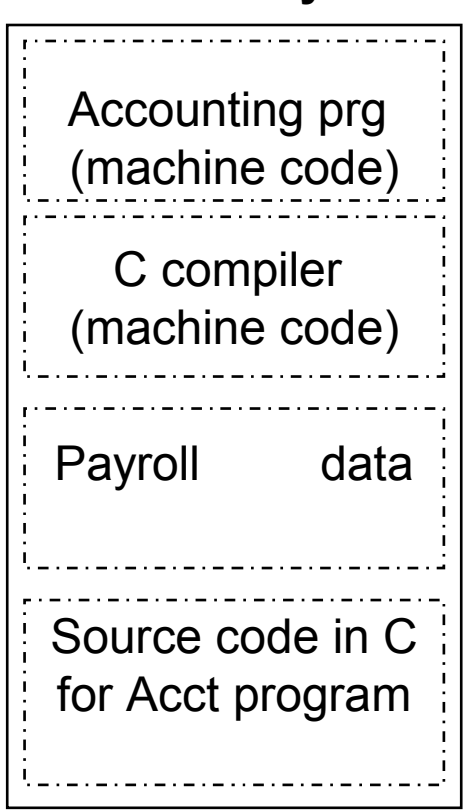
Instruction Set Architecture (ISA)

- ISA, or simply architecture – the abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, I/O, …
  - Enables implementations of varying cost and performance to run identical software

- The combination of the basic (user portion of the) instruction set (the ISA) and the operating system interface is called the application binary interface (ABI)
  - Defines a standard for binary portability across computers.

Two Key Principles of Machine Design

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 (von Neumann) 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 – leads industry to align around a small number of ISAs

Assembly Language Instructions

- The language of the machine
  - Want an ISA that makes it easy to build the hardware *and* the compiler (whose job it is to translate programs written in a high level language (like C) to assembly code) while maximizing performance and minimizing cost
- Our target: the MIPS ISA
  - similar to other ISAs developed since the 1980's
  - used by Broadcom, Cisco, NEC, Nintendo, Sony, …

Design goals: maximize performance, minimize cost, reduce design time (time-to-market), minimize power consumption, maximize reliability

RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited number of addressing modes
  - limited number of operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC …
- Instruction sets are measured by how well compilers can use them as opposed to how well assembly language programmers can use them
- CISC (C for complex), e.g., Intel x86

The Four RISC Design Principles

1. Simplicity favors regularity.
2. Smaller is faster.
3. Make the common case fast.
4. Good design demands good compromises.

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
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction specifies exactly three operands

destination ← source1 op source2

  - Operand order is fixed (the destination is specified first)
- The operands are contained in the datapath's register file (`$t0, $s1, $s2`)

MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction specifies exactly three operands

destination ← source1 op source2

  - Operand order is fixed (the destination is specified first)
- The operands are contained in the datapath's register file (`$t0, $s1, $s2`)

Compiling More Complex Statements

- Assuming variable b is stored in register `$s1`, c is stored in `$s2`, and d is stored in `$s3` and the result is to be left in `$s0`, what is the assembler equivalent to the C statement

```
h = (b - c) + d
```

Compiling More Complex Statements

- Assuming variable b is stored in register `$s1`, c is stored in `$s2`, and d is stored in `$s3` and the result is to be left in `$s0`, what is the assembler equivalent to the C statement

```
h = (b - c) + d
```

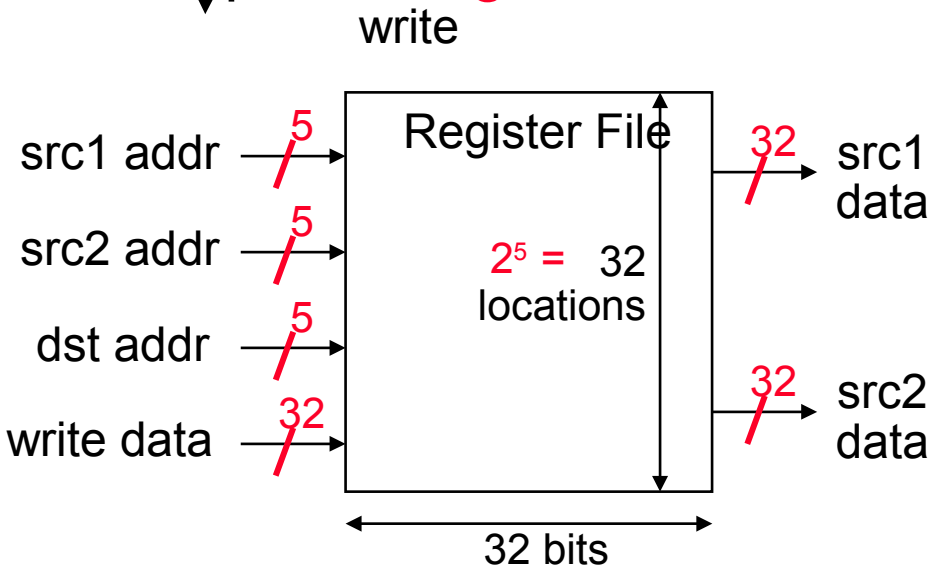
```
sub  $t0, $s1, $s2
add  $s0, $t0, $s3
```

MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Thirty-two 32-bit registers
    - Two read ports
    - One write port
- Registers are
  - Faster than main memory
    - Smaller is faster & Make the common case fast
  - Easy for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order
  - Improves code density
    - Since register are named with fewer bits than a memory location
- Register addresses are indicated by using $

MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Thirty-two 32-bit registers
    - Two read ports
    - One write port



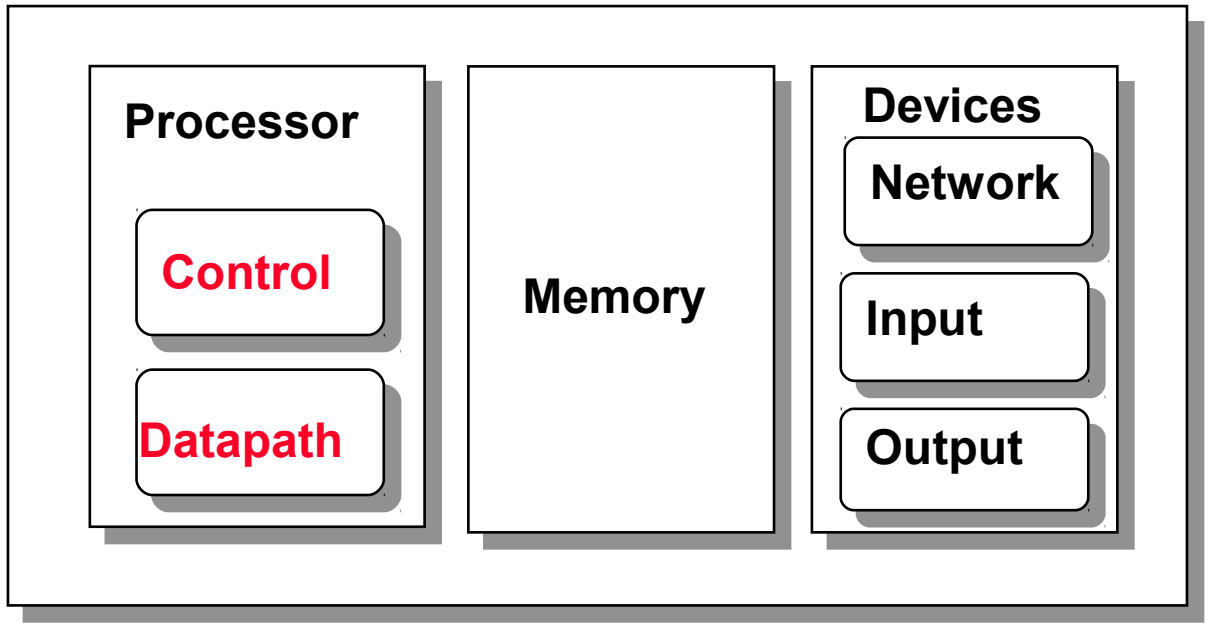
- Registers are
  - Faster than main memory
    - Smaller is faster & Make the common case fast
  - Easy for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order
  - Improves code density
    - Since register are named with fewer bits than a memory location
- Register addresses are indicated by using $

MIPS Register Naming Convention

Nick Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k0 - $k1	26-27	reserved for OS	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

Registers vs. Memory

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

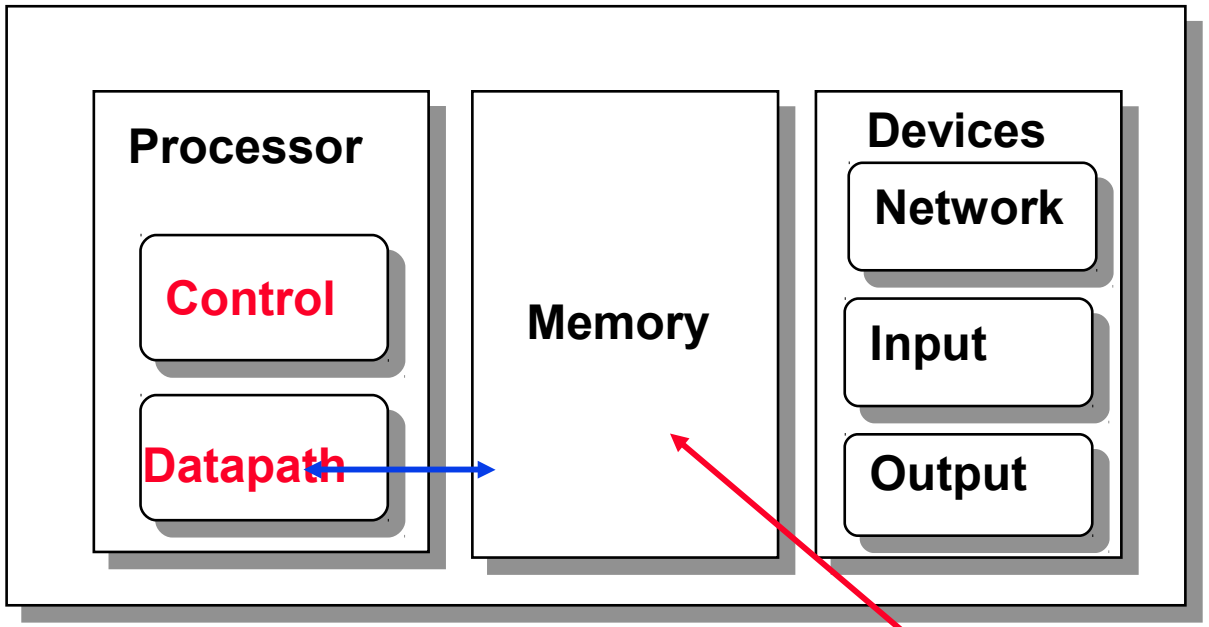


- The compiler associates variables with registers

What about programs with lots of variables?

Registers vs. Memory

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

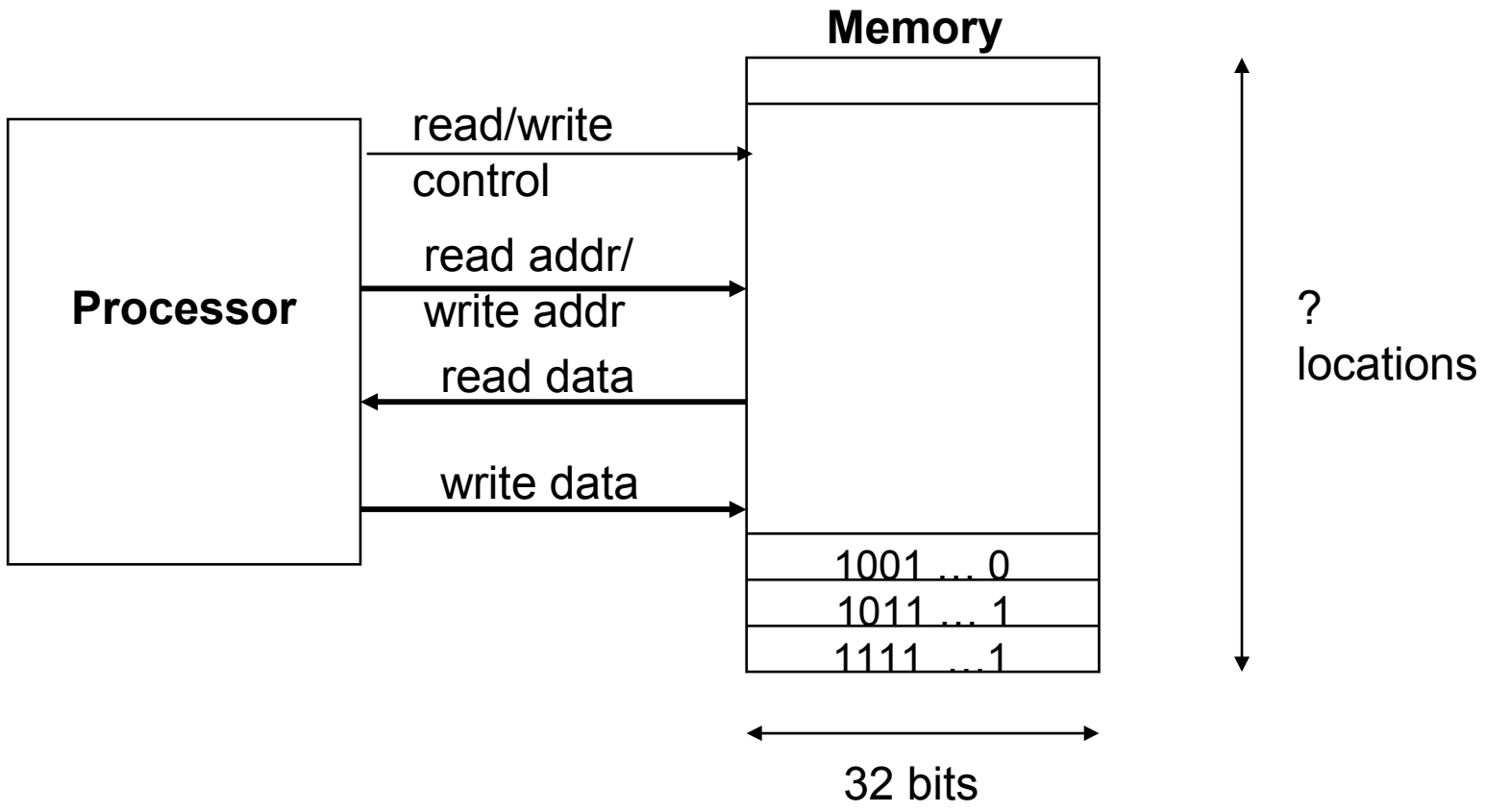


- The compiler associates variables with registers

What about programs with lots of variables?

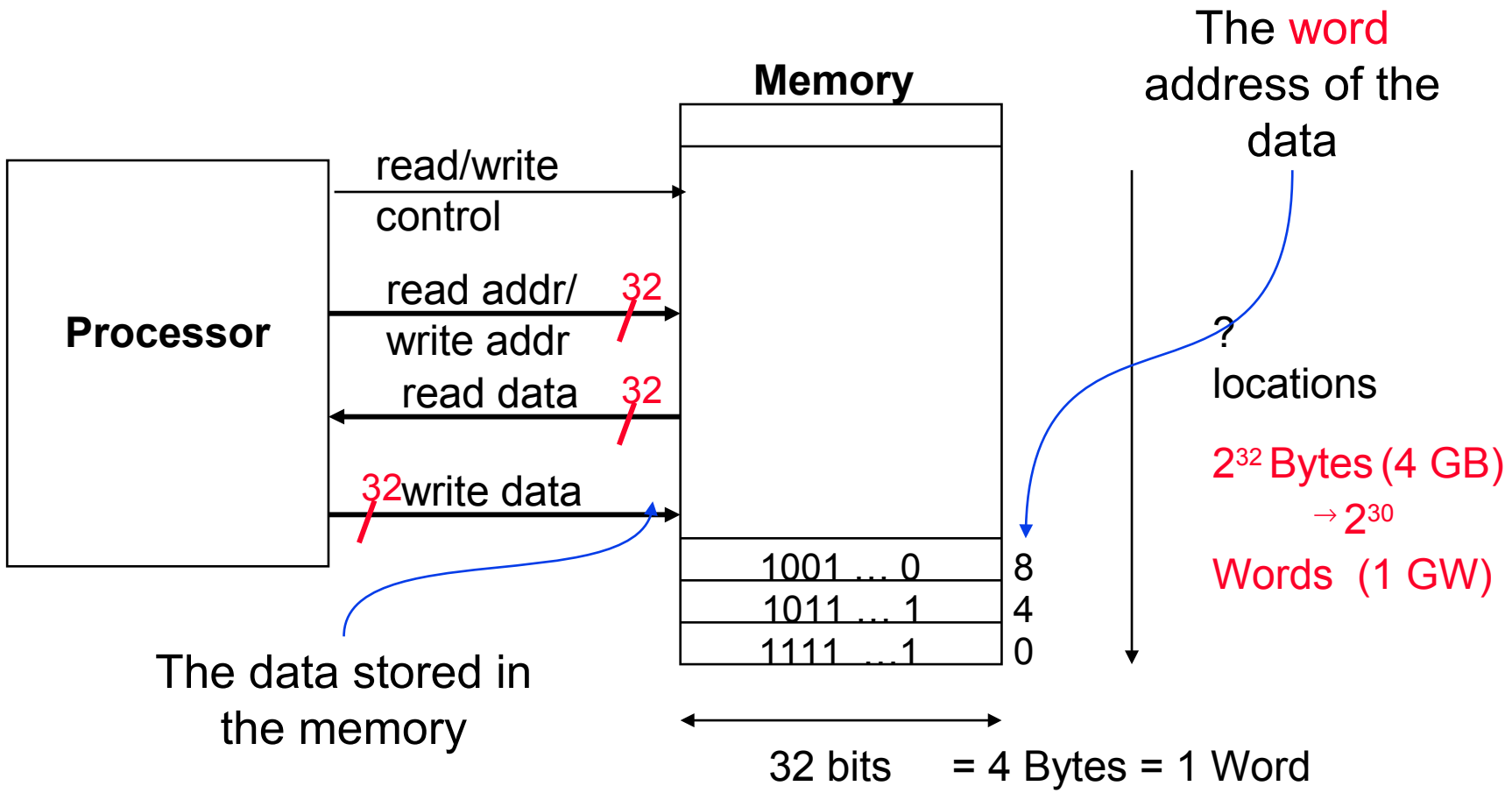
Processor – Memory Interconnections

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



Processor – Memory Interconnections

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

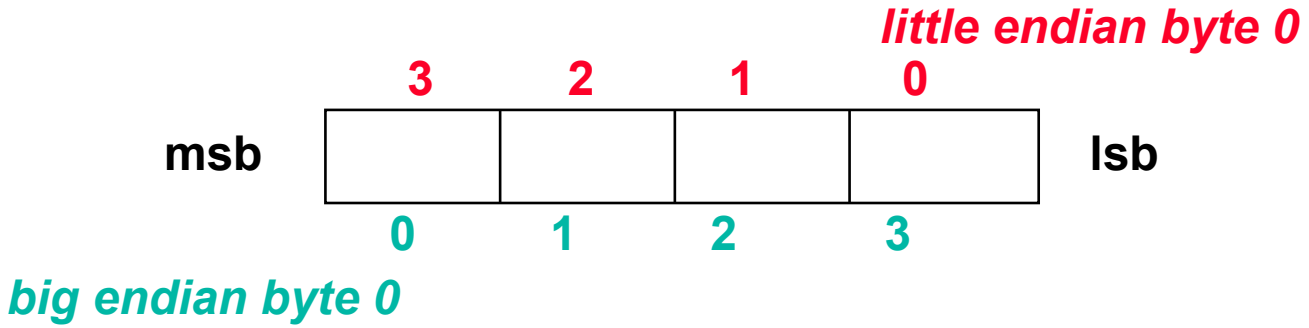


Word Addresses vs Byte Addresses

- Alignment restriction - the memory address of a word *must* be on natural word boundaries (a multiple of 4 in MIPS-32)
- But since 8-bit bytes are so useful, most architectures also support addressing individual bytes in memory
- Big Endian: least-significant byte is word address

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

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



Accessing Memory

- MIPS has two basic data transfer instructions for accessing memory (assume `$s3` holds 24_{10})

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

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

- The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – memory address
  - where in the register file to write to (load) or read from (store) – register destination (source)
- The memory address is formed by summing the constant portion of the instruction and the contents of the second register

Accessing Memory

- MIPS has two basic data transfer instructions for accessing memory (assume `$s3` holds 24_{10})

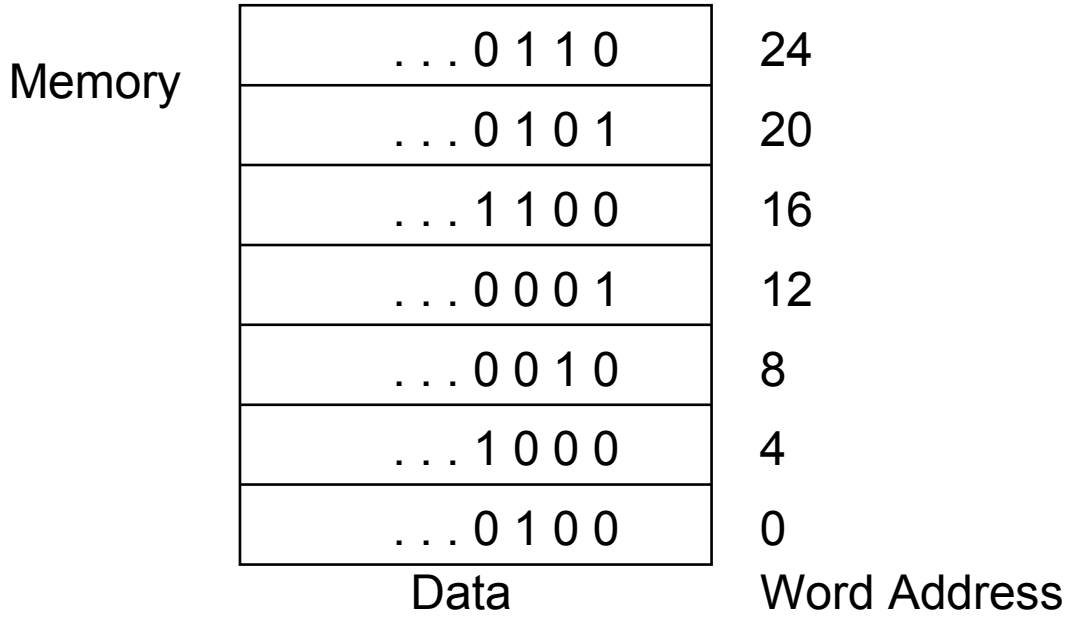
```
lw   $t0, 4($s3)   #load word from memory
                   28(10)
sw   $t0, 8($s3)   #store word to  memory
                   32(10)
```

- The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – memory address
  - where in the register file to write to (load) or read from (store) – register destination (source)
- The memory address is formed by summing the constant portion of the instruction and the contents of the second register

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

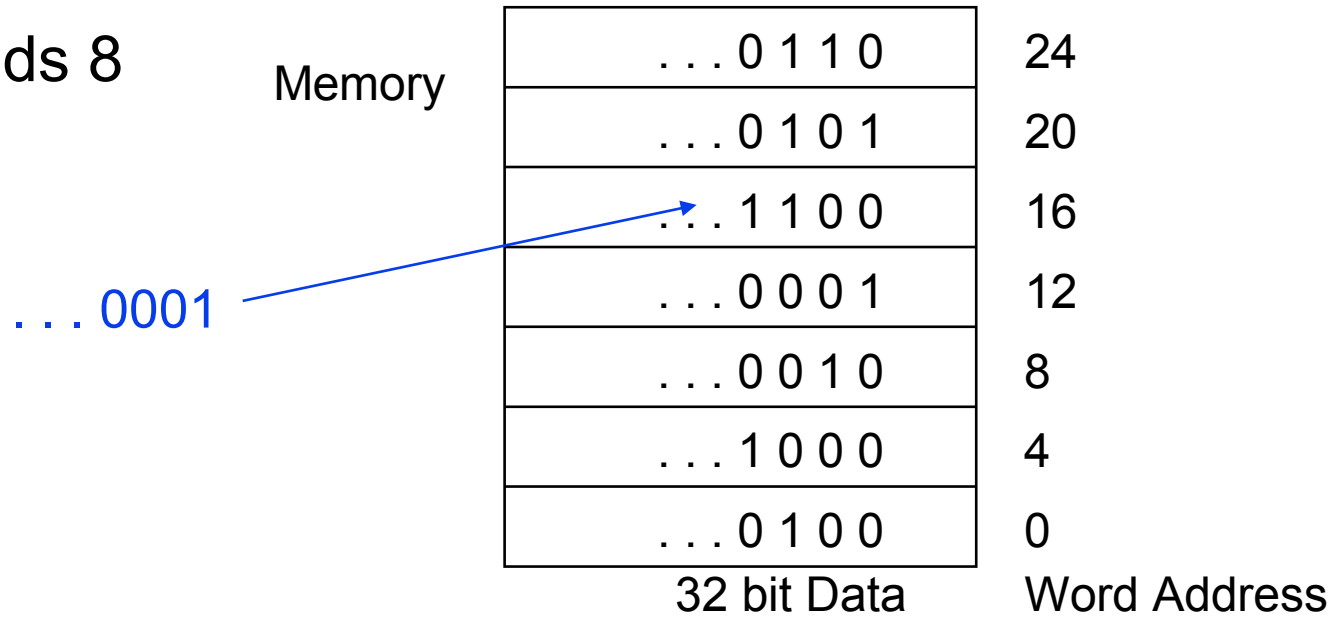



```
lw    $t0, 4($s3)   #what? is loaded into $t0

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

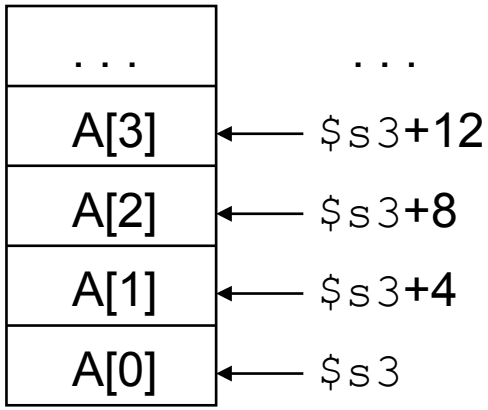
```
sw    $t0, 8($s3)   #$t0 is stored where?
```

in memory 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 three statement MIPS assembly code for the C statement

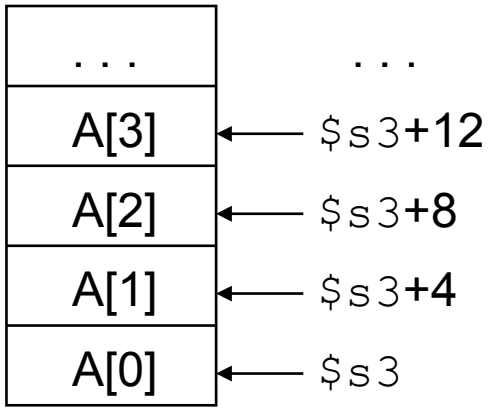
```
A[8] = A[2] - b
```



Compiling with Loads and Stores

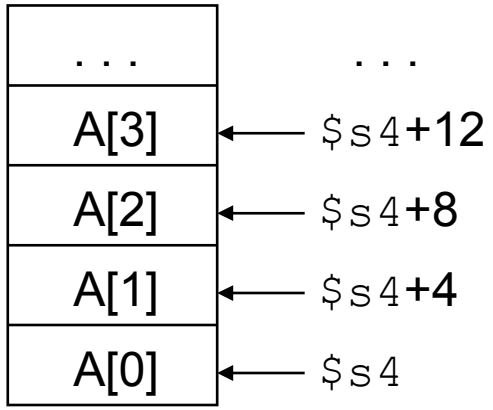
- Assuming variable `b` is stored in `$s2` and that the base address of array A is in `$s3`, what is the three statement MIPS assembly code for the C statement

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




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

Compiling with a Variable Array Index

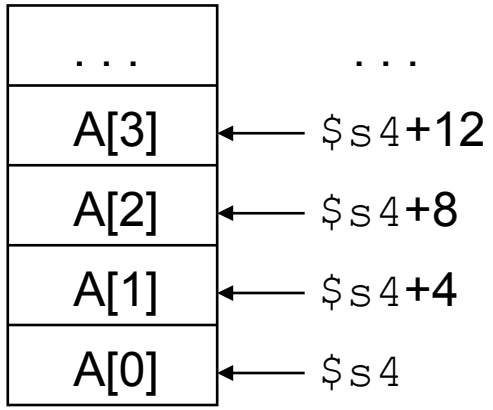


- Assuming that the base address of array A is in register `$s4`, and variables b, c, and i are in `$s1`, `$s2`, and `$s3`, respectively, complete the MIPS assembly code for the C statement

```
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, complete 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
```

Dealing with Constants

- Small constants are used quite frequently (50% of operands in many common programs)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- Solutions? Why not?
  - Create hard-wired registers (like $zero) for constants like 1, 2, 4, 10, …
  - Put “typical constants” in memory and load them
  - . . .

- How do we make this work? How do we Make the common case fast !

Constant (or Immediate) Operands

- Include constants inside arithmetic instructions
  - Much faster than if they have to be loaded from memory (they come in from memory *with* the instruction itself)
- MIPS immediate instructions

```
addi $s3, $s3, 4     #$s3 = $s3 + 4
```

- And to move (copy) the contents of one register to another in *one* instruction

```
add  $s3, $s2, $zero
```

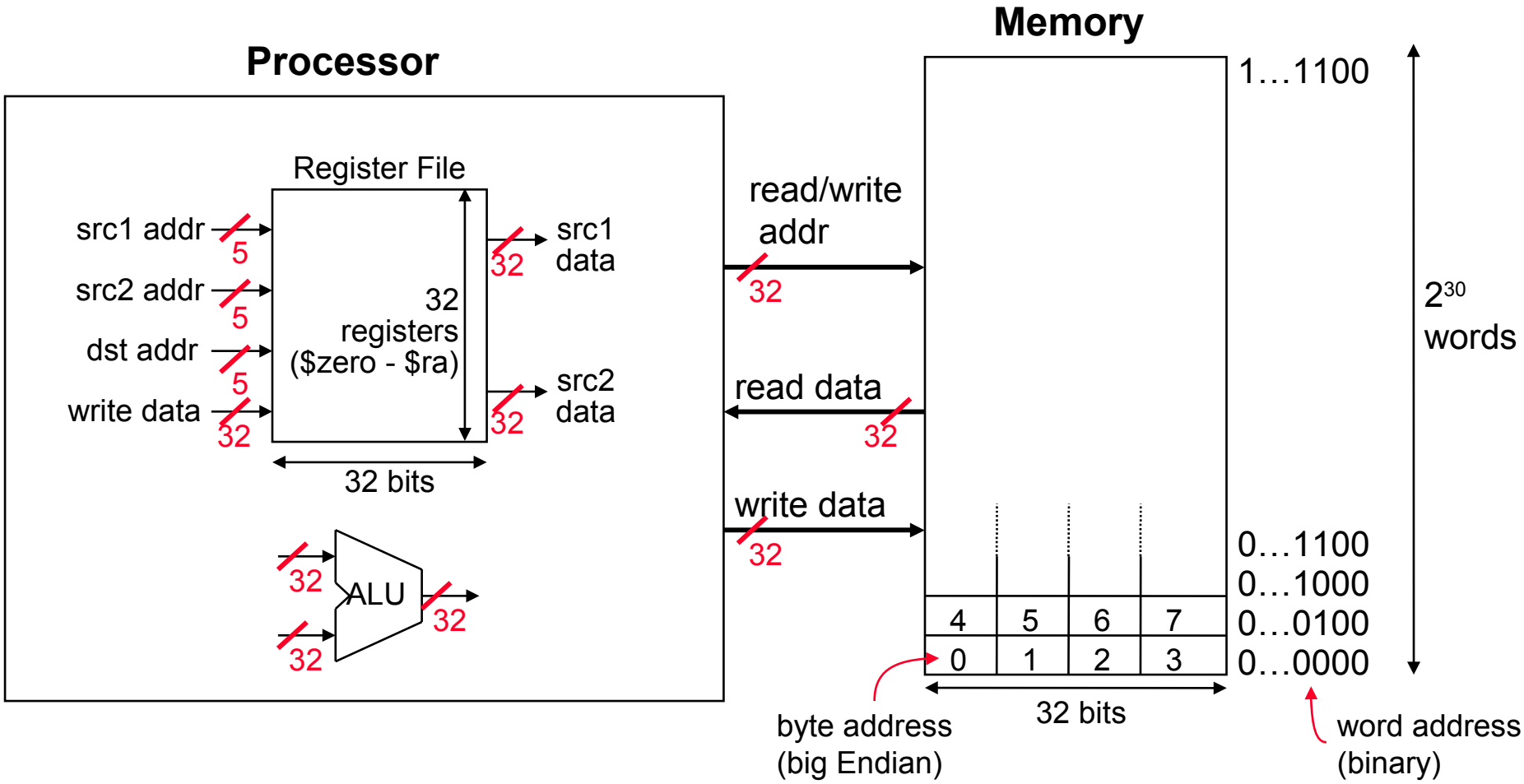
There is no `subi` *instruction, can you guess why not?*

MIPS Instructions, so far

Category	Instr	Example	Meaning
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	addi $s1, $s2, 4	$s1 = $s2 + 4
Data transfer	load word	lw $s1, 32($s2)	$s1 = Memory($s2+32)
	store word	sw $s1, 32($s2)	Memory($s2+32) = $s1

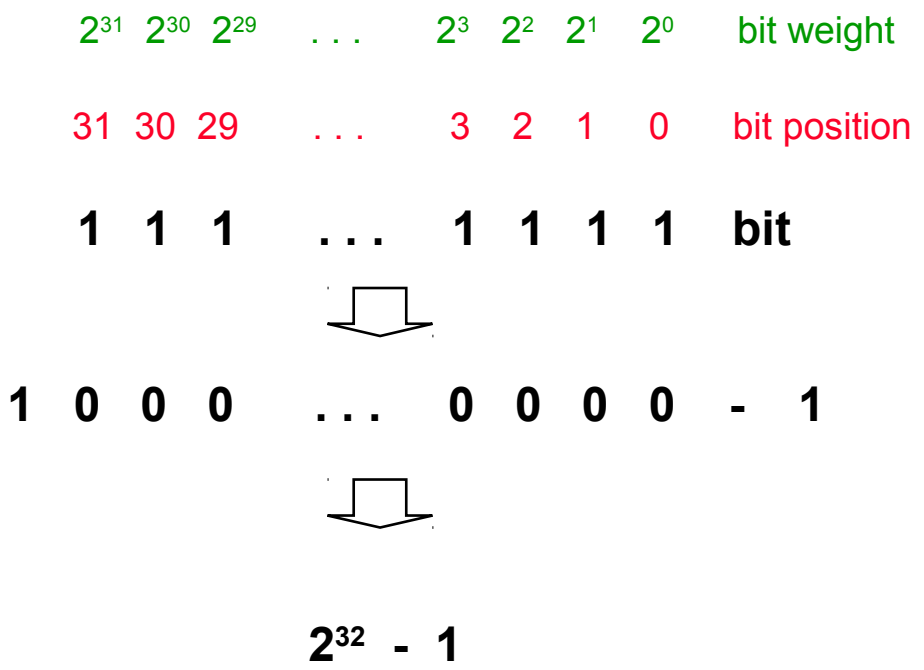
MIPS Organization, so far

- Arithmetic instructions – to/from the register file
- Load/store instructions – to/from memory



Review: Unsigned Binary Representation

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	$2^{32} - 1$

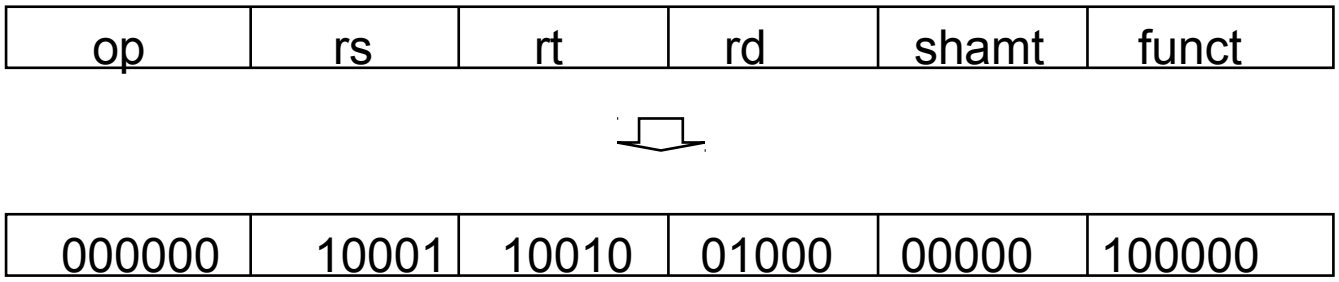


Machine Language - Arithmetic Instruction

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`

  registers have numbers `$t0=$8,$s1=$17,$s2=$18`

- Instruction Format:



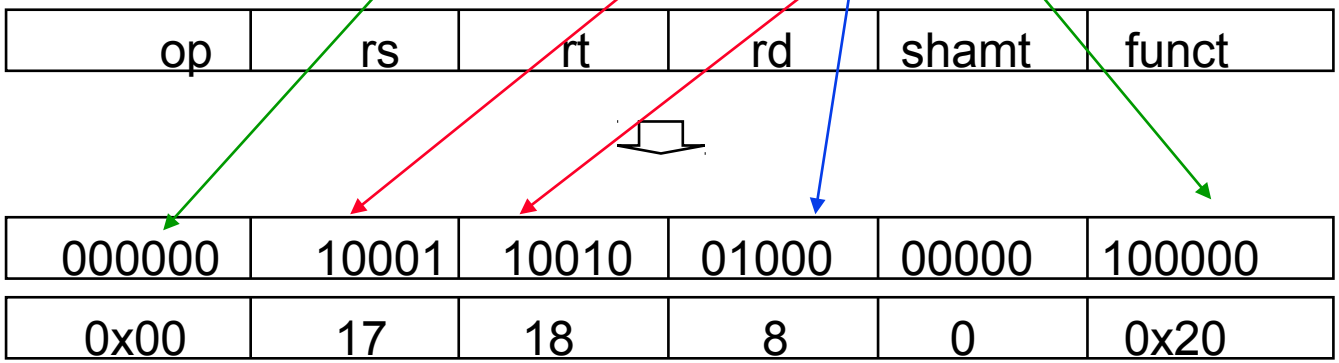
Can you guess what the field names stand for?

Machine Language - Arithmetic Instruction

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`

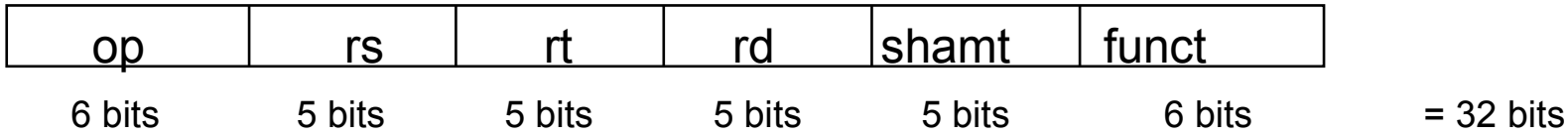
  registers have numbers `$t0=$8, $s1=$17, $s2=$18`

- Instruction Format:



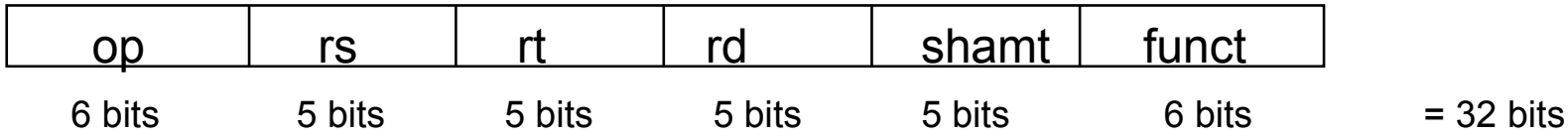
Can you guess what the field names stand for?

MIPS Instruction Fields



- *op*
- *rs*
- *rt*
- *rd*
- *shamt*
- *funct*

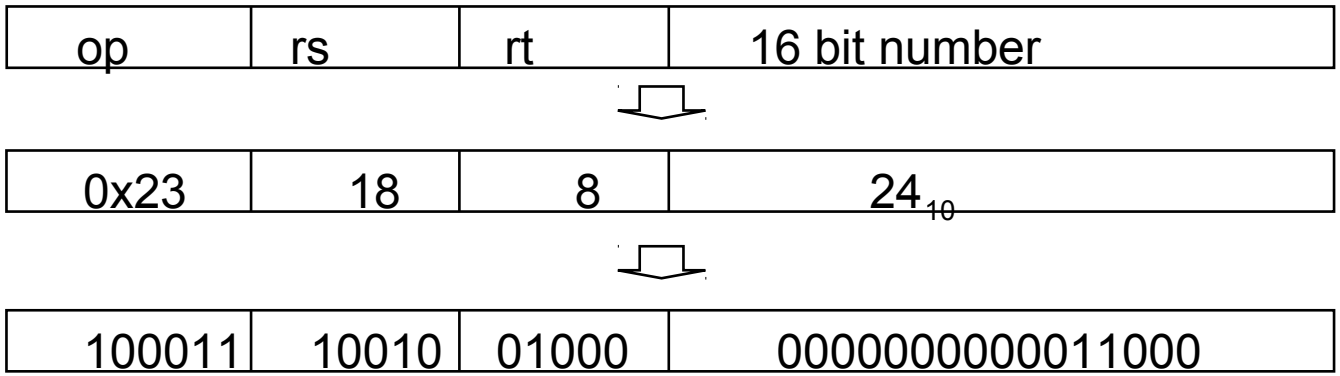
MIPS Instruction Fields



- *op* **op**code indicating operation to be performed
- *rs* **r**egister file address of the first **s**ource operand
- *rt* **r**egister file address of the second source operand
- *rd* **r**egister file address of the result's **d**estination
- *shamt* **sh**ift **am**oun**t** (for shift instructions)
- *funct* **funct**ion code that selects the specific variant of the operation specified in the opcode field

Machine Language - Load Instruction

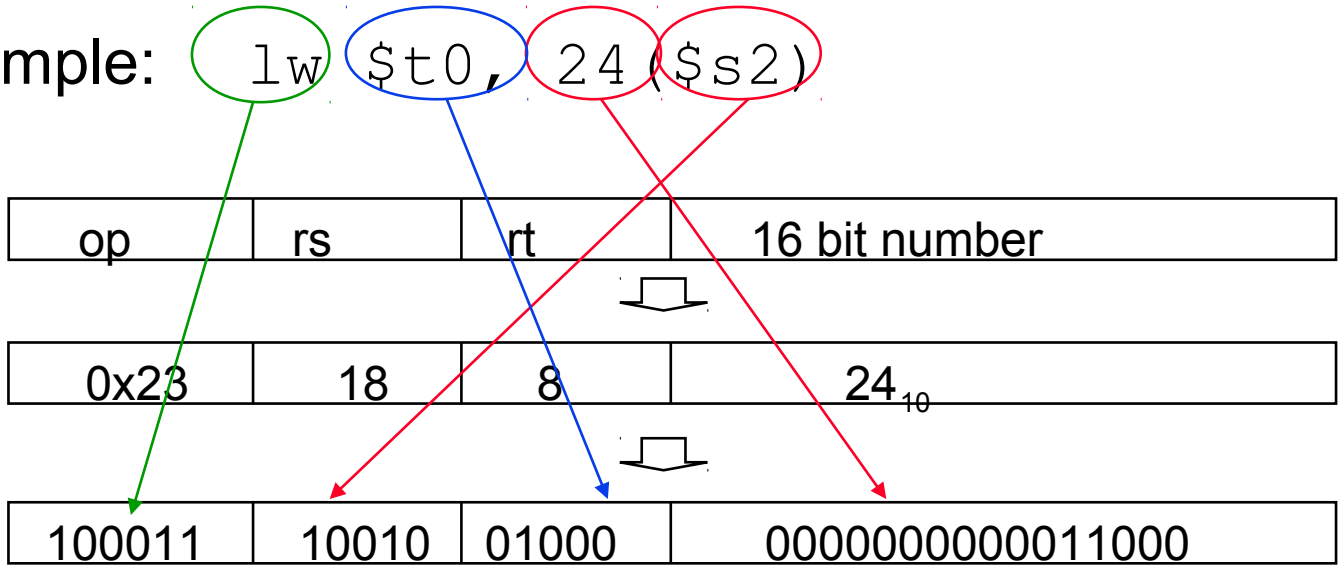
- Consider the load-word and store-word instr's
  - What would the regularity principle have us do?
    - But . . . Good design demands compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions (previous format was R-type for register)
- Example: `lw $t0, 24($s2)`



Where's the compromise?

Machine Language - Load Instruction

- Consider the load-word and store-word instr's
  - What would the regularity principle have us do?
    - But . . . Good design demands compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions (previous format was R-type for register)
- Example: `lw $t0, 24($s2)`

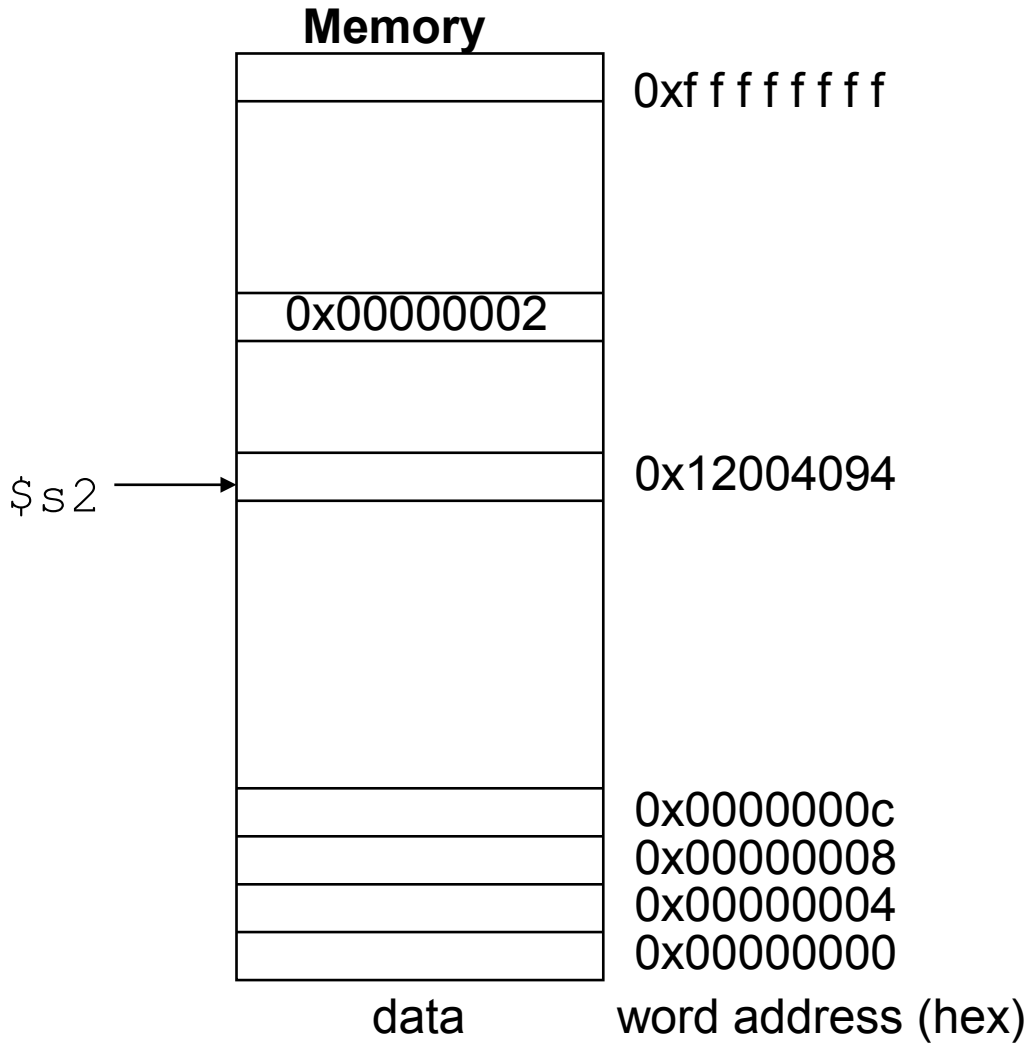


Where's the compromise?

Memory Address Location

- Example: `lw $t0, 24($s2)`

24_{10} + $s2 =



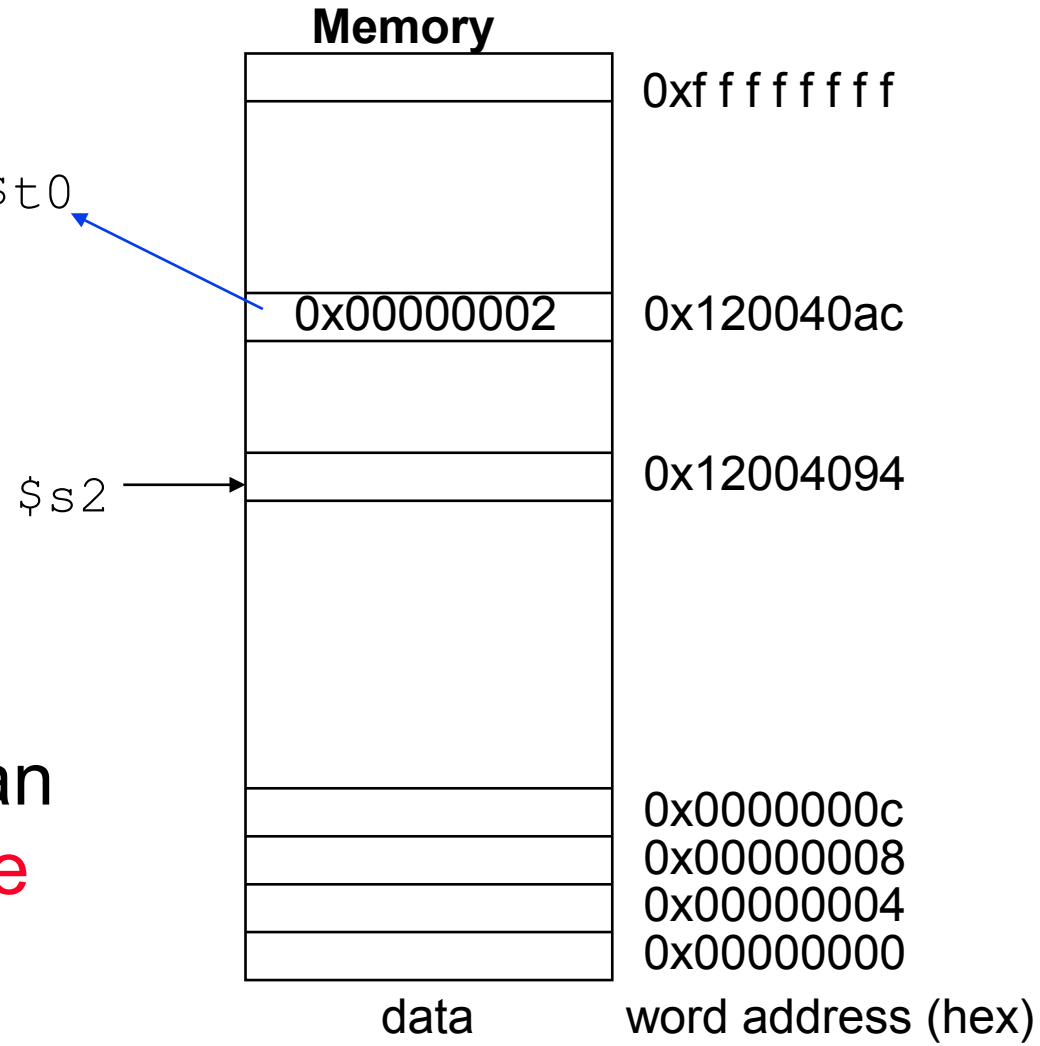
Memory Address Location

- Example: `lw $t0, 24($s2)`

24_{10} + $s2 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac

Note that the offset can be positive or negative



Review: Signed Binary Representation

	2'sc 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

complement all the bits

0101

and add a 1

0110

1011

and add a 1

1010

complement all the bits

Machine Language - Store Instruction

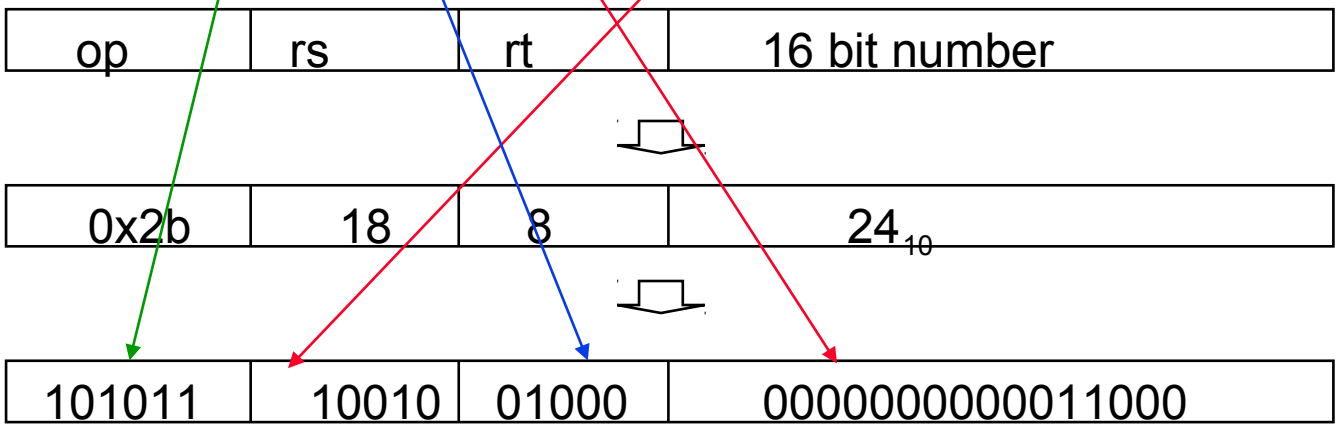
- Example: `sw $t0, 24($s2)`

op	rs	rt	16 bit number
0x2b	18	8	24_{10}
101011	10010	01000	0000000000011000

- A 16-bit offset means access is limited to memory locations within a range of $+2^{13}-1$ to -2^{13} (~8,192) words ($+2^{15}-1$ to -2^{15} (~32,768) bytes) of the address in the base register `$s2`
  - 2's complement (1 sign bit + 15 magnitude bits)

Machine Language - Store Instruction

- Example: `sw $t0, 24($s2)`



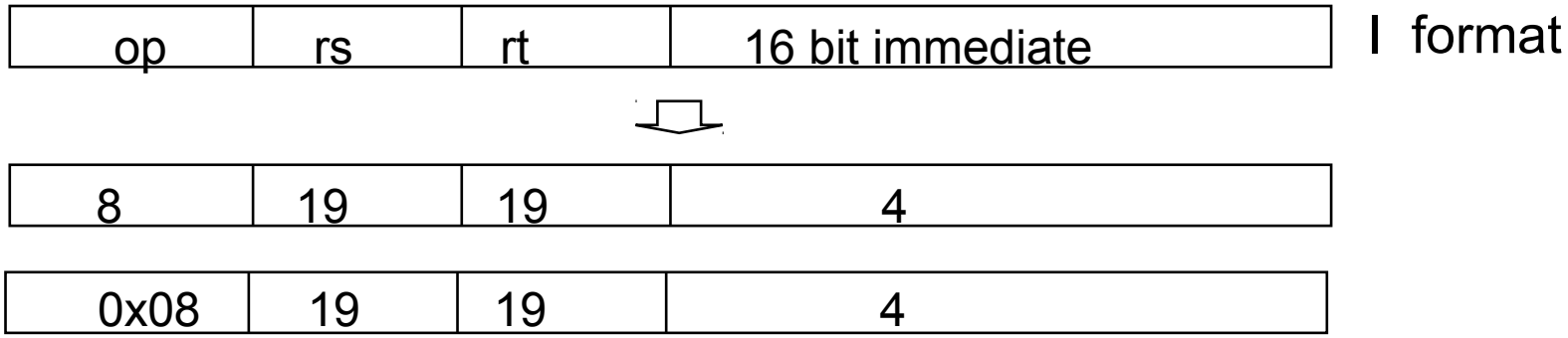
- A 16-bit offset means access is limited to memory locations within a range of $+2^{13}-1$ to -2^{13} (~8,192) words ($+2^{15}-1$ to -2^{15} (~32,768) bytes) of the address in the base register `$s2`
  - 2's complement (1 sign bit + 15 magnitude bits)

Machine Language – Immediate Instructions

- What instruction format is used for the `addi` ?

```
addi $s3, $s3, 4     #$s3 = $s3 + 4
```

- Machine format:



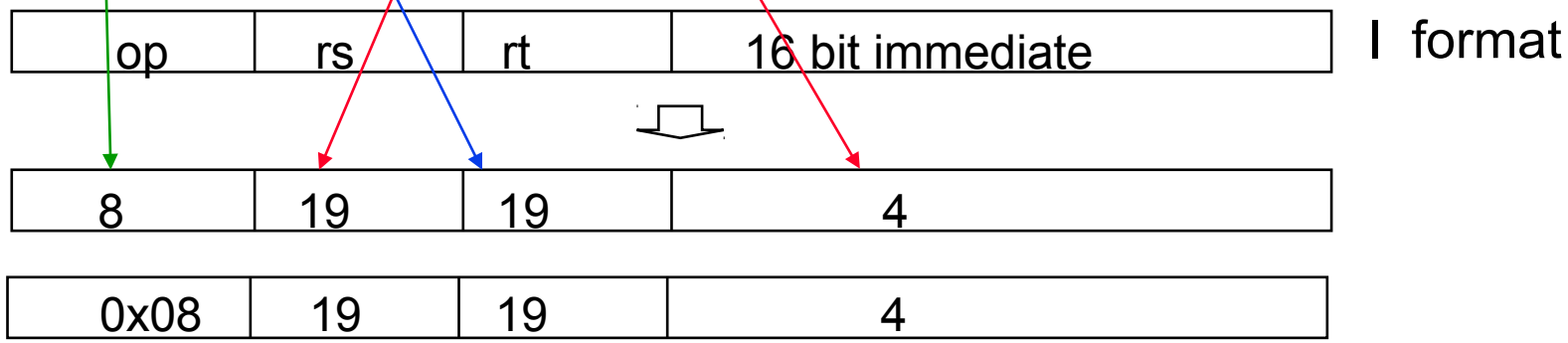
- The constant is kept inside the instruction itself!
  - So must use the I format – Immediate format
  - Limits immediate values to the range $+2^{15}-1$ to -2^{15}

Machine Language – Immediate Instructions

- What instruction format is used for the `addi` ?

```
addi $s3, $s3, 4     #$s3 = $s3 + 4
```

- Machine format:



- The constant is kept inside the instruction itself!
  - So must use the I format – Immediate format
  - Limits immediate values to the range $+2^{15}-1$ to -2^{15}

Instruction Format Encoding, so far

- Can reduce the complexity with multiple formats by keeping them as similar as possible
  - First three fields are the same in R-type and I-type
- Each format has a distinct set of values in the op field

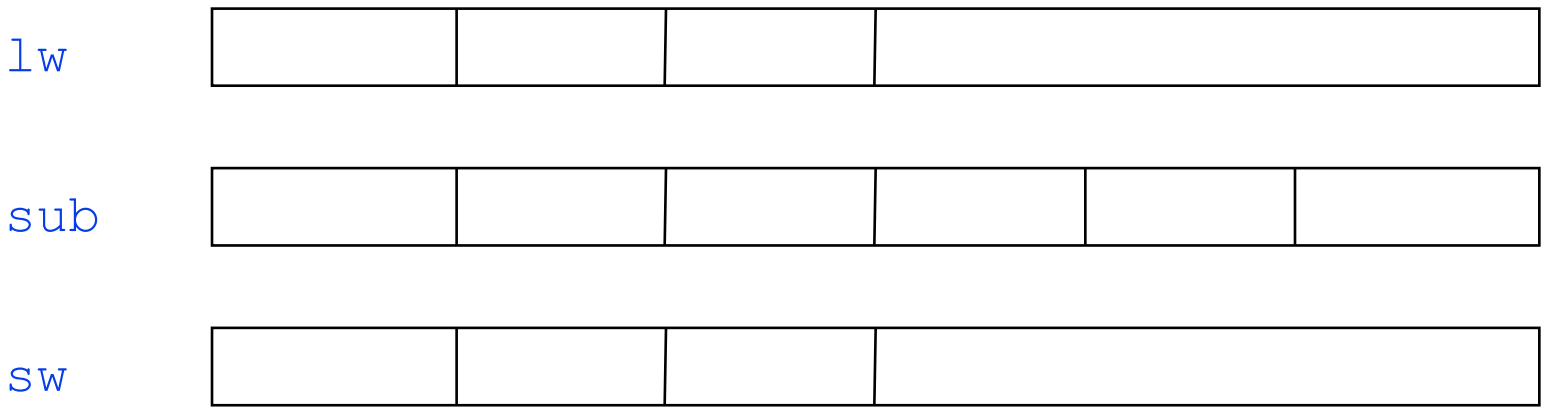
Instr	Frmt	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	NA
`sub`	R	0	reg	reg	reg	0	34_{ten}	NA
`addi`	I	8_{ten}	reg	reg	NA	NA	NA	constant
`lw`	I	35_{ten}	reg	reg	NA	NA	NA	address
`sw`	I	43_{ten}	reg	reg	NA	NA	NA	address

Assembling Code

- Remember the assembler code we compiled last lecture for the C statement

```
                    A[8] = A[2] - b
lw    $t0, 8($s3)     #load A[2] into $t0
sub   $t0, $t0, $s2   #subtract b from A[2]
sw    $t0, 32($s3)    #store result in A[8]
```

- Assemble the MIPS object code for these three instructions (in decimal is fine)

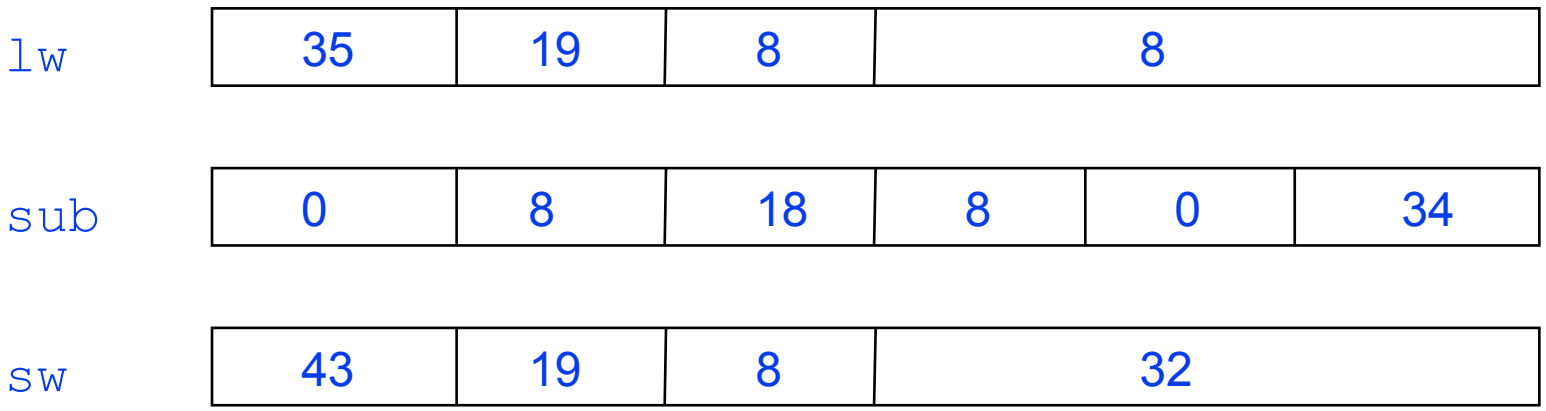


Assembling Code

- Remember the assembler code we compiled last lecture for the C statement

```
                    A[8] = A[2] - b
lw    $t0, 8($s3)     #load A[2] into $t0
sub   $t0, $t0, $s2   #subtract b from A[2]
sw    $t0, 32($s3)    #store result in A[8]
```

- Assemble the MIPS object code for these three instructions (in decimal is fine)

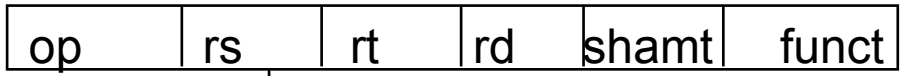


Review: MIPS Instructions, so far

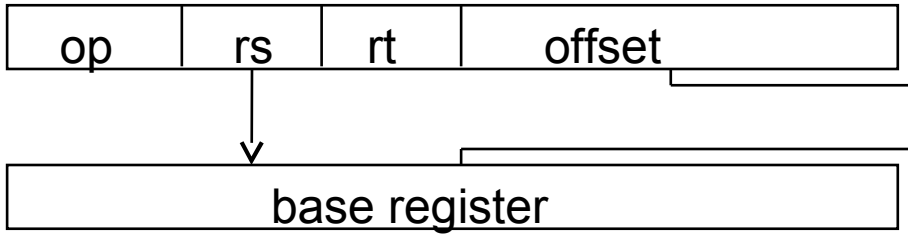
Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	add	0 & 20_{hex}	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 & 22_{hex}	sub $s1, $s2, $s3	$s1 = $s2 - $s3
Arithmetic (I format)	add immediate	8_{hex}	addi $s1, $s2, 4	$s1 = $s2 + 4
Data transfer (I format)	load word	23_{hex}	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	$2b_{hex}$	sw $s1, 100($s2)	Memory($s2+100) = $s1

Review: Addressing Modes, so far

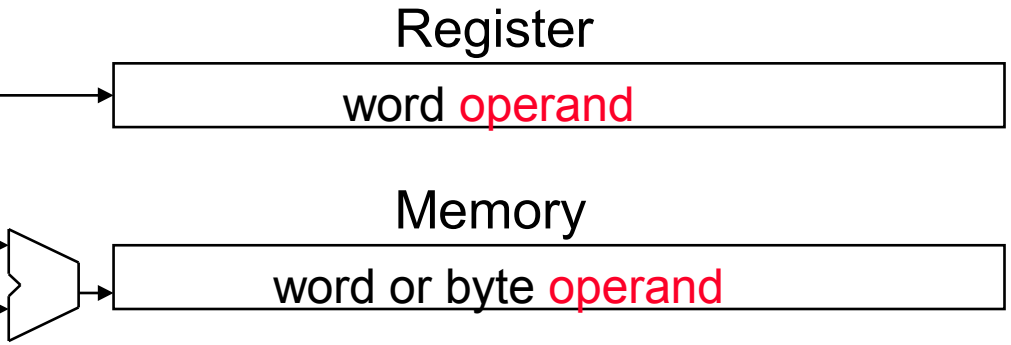
1. Register addressing



2. Base (displacement) addressing

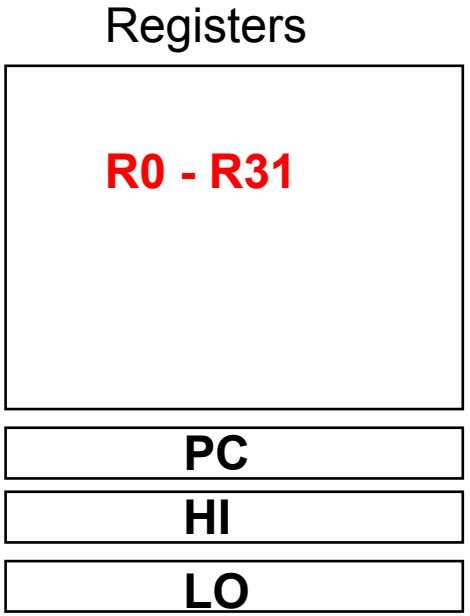


3. Immediate addressing



MIPS32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide