Computer architecture

MIPS Instruction Set

Introduction

- **Objective: To introduce the MIPS instruction set and to show how MIPS instructions are represented in the computer.**
- **The stored-program concept:**
  - **Instructions are represented as numbers.**
  - **Programs can be stored in memory to be read or written just like data.**

MIPS Instruction Set

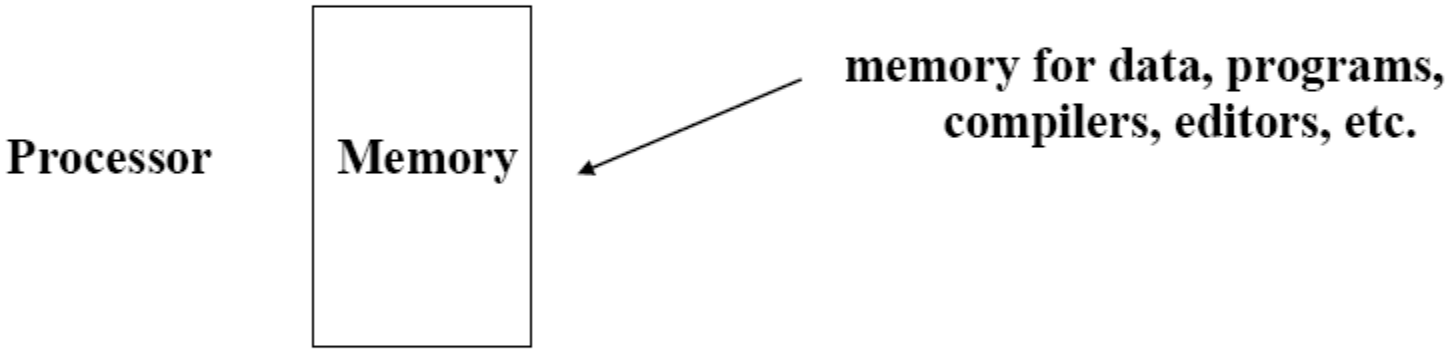
MIPS (originally an acronym for Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies (formerly MIPS Computer Systems, Inc.). The early MIPS architectures were 32-bit, with 64-bit versions added later.

After MIPS, several key technologies and architectures emerged:

1. **ARM Architecture: Dominates mobile and embedded systems due to its energy efficiency and flexibility, with companies like Apple and Qualcomm using ARM-based chips.**
2. **x86 Architecture (Intel & AMD): Continues to lead in desktops, servers, and high-performance computing, especially with innovations from Intel and AMD.**
3. **RISC-V: An open-source RISC architecture gaining popularity due to its customizability and lack of licensing fees, particularly in embedded systems and IoT.**
4. **Apple Silicon: Apple's custom ARM-based processors (M1, M2) offer high performance and power efficiency, marking a shift from x86.**
5. **AI-Specific Architectures: Specialized processors like Google's TPU and NVIDIA's GPUs are designed to accelerate AI and machine learning tasks.**
6. **FPGAs and ASICs: Custom silicon solutions are used for specialized tasks in fields like encryption and AI.**
7. **Quantum Computing: A futuristic technology that could eventually surpass classical computing in certain problem-solving domains.**
8. **Multi-core and Hybrid Architectures: Combining high-performance and low-power cores for greater efficiency in modern processors.**

Stored Program Concept

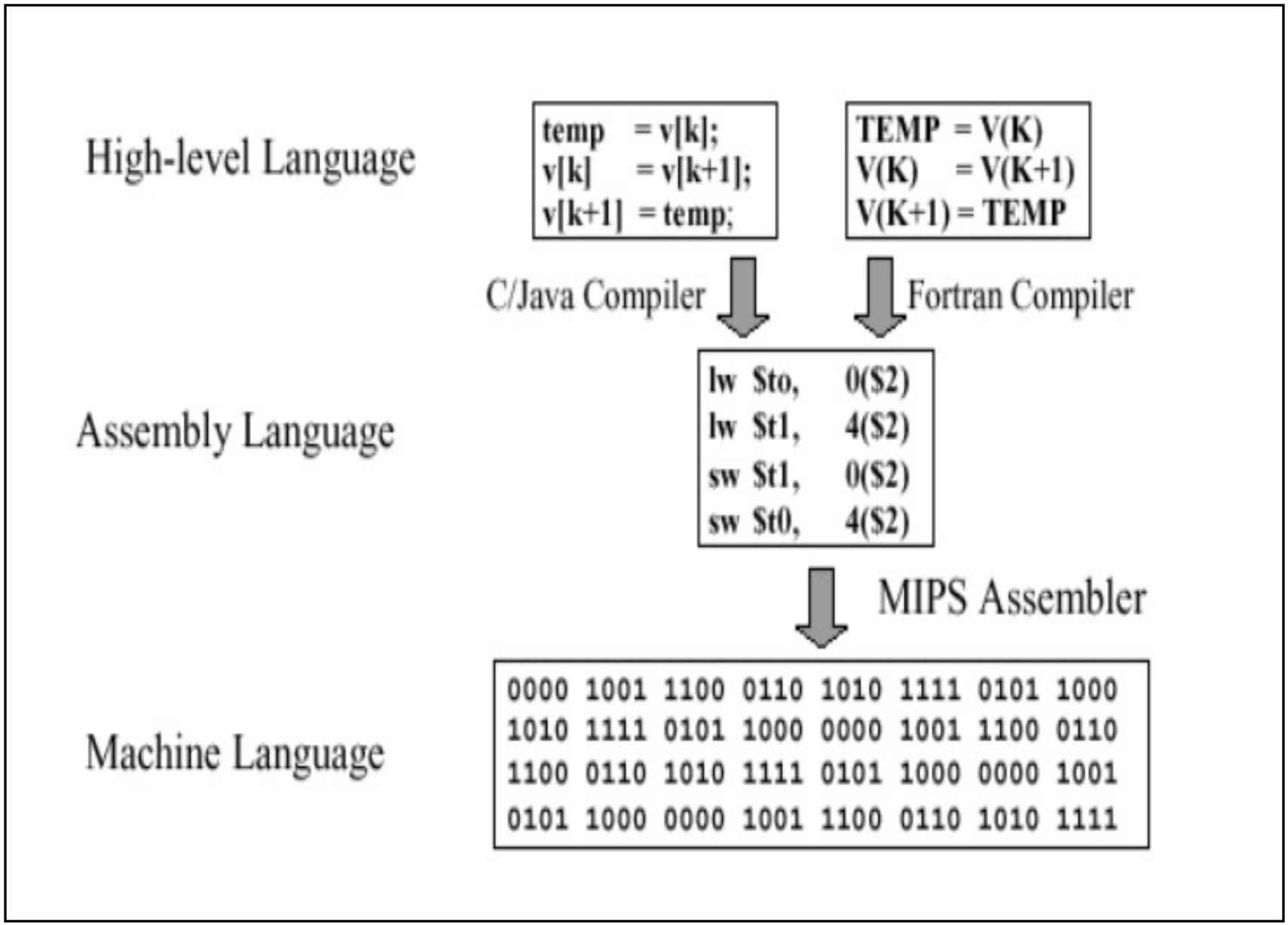
- Instructions are just a sequence of 32 bits
- Programs are stored in memory
  — to be read or written just like data



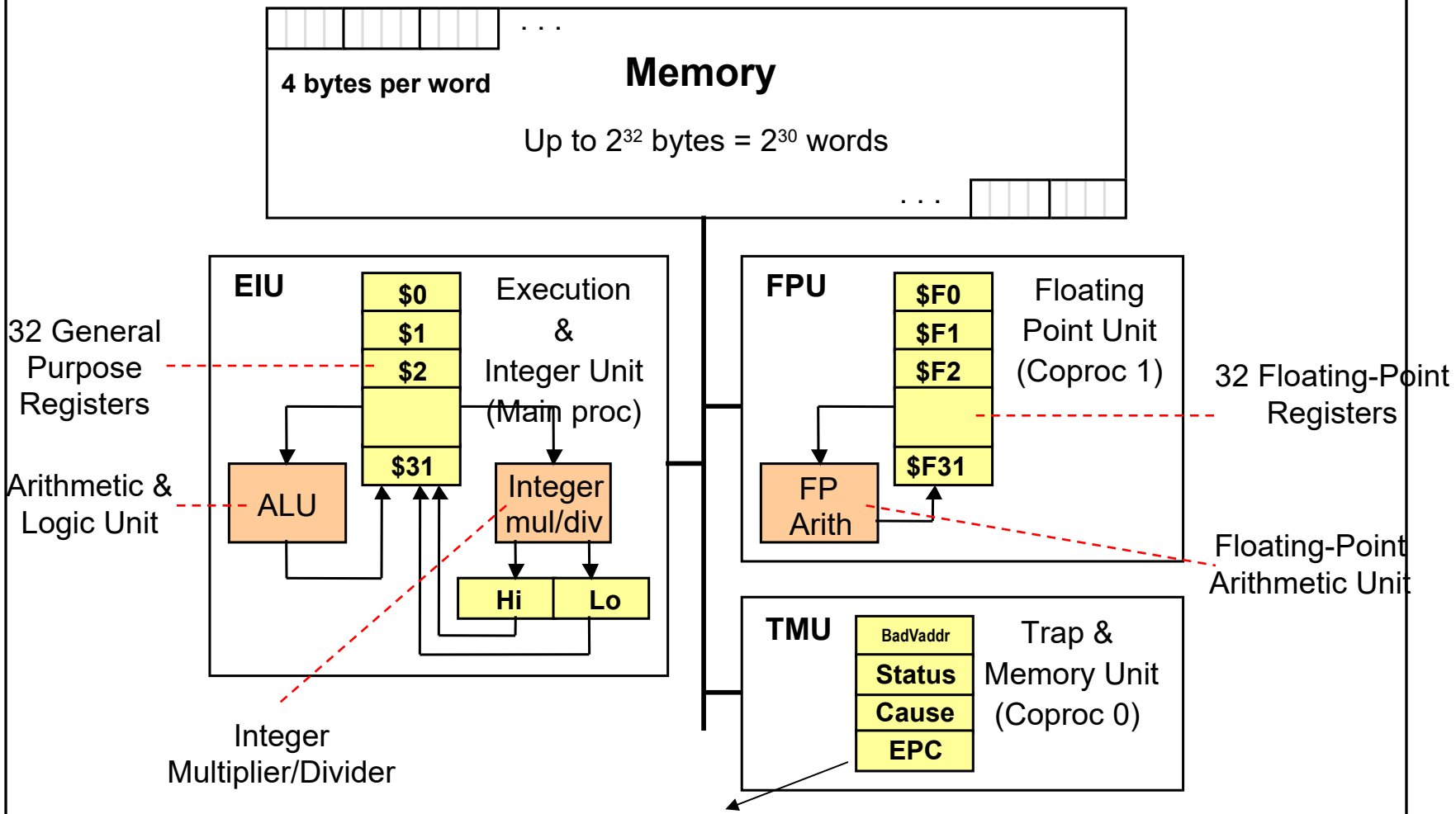
- Fetch & Execute Cycle
  - Instruction is fetched and put into a special register
  - Bits in the instruction register determine the subsequent actions
  - When done, fetch the next instruction and continue execution

Instructions

- **Language of the machine**
- **More primitive than higher level languages (e.g. C, C++, Java)**
  - **e.g. no sophisticated control flow , primitive data structures**
- **MIPS – ISA developed in the early 80's (RISC)**
  - **Similar to other RISC architectures developed since the 1980's**
  - **Almost 100 million MIPS processors manufactured in 2002**
  - **Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …**
  - **Regular (32 bit instructions, small number of different formats)**
  - **Relatively small number of instructions**
  - **Register architecture (all instructions operate on registers)**
  - **Load/Store architecture (memory accessed only with load/store instructions, with few addressing modes)**
- ***Design goals: maximize performance and minimize cost, reduce processor design time***

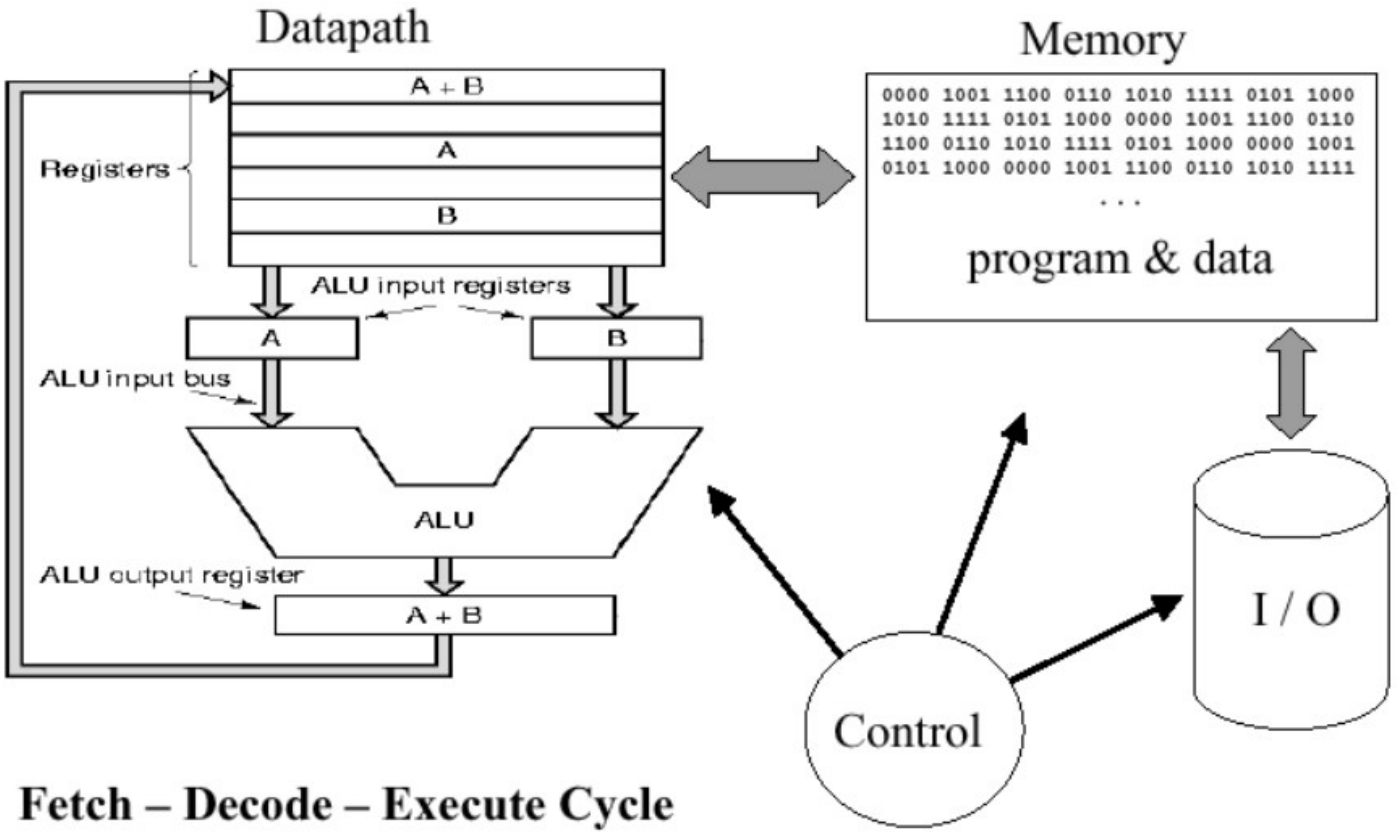

High-level Language
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
TEMP = V(K)
V(K) = V(K+1)
V(K+1) = TEMP
C/Java Compiler
Fortran Compiler
Assembly Language
lw $to, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
MIPS Assembler
Machine Language
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

Logical View of the MIPS Processor



Exception Program Counter (or **Exception PC**). The EPC is a special register that holds the address of the instruction that caused an exception or trap in the processor.

Fetch - Decode - Execute Cycle



Fetch – Decode – Execute Cycle

machine device shipped with MIPS cores

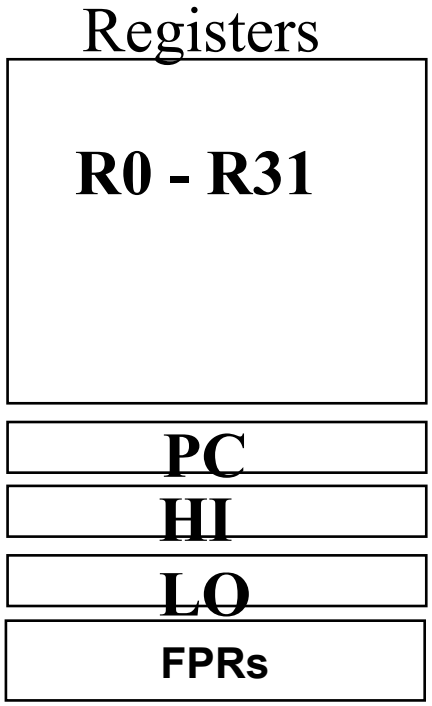
- Cable Modems 94%
- DSL Modems 40%
- VDSL Modems 93%
- IDTV 40%
- Cable STBs 76%
- DVD Recorder 75%
- Game Consoles 76%
- Office Automation 48%
- Color Laser Printers 62%
- Commercial Color Copiers 73%

- Source: Website of MIPS Technologies, Inc., 2004.

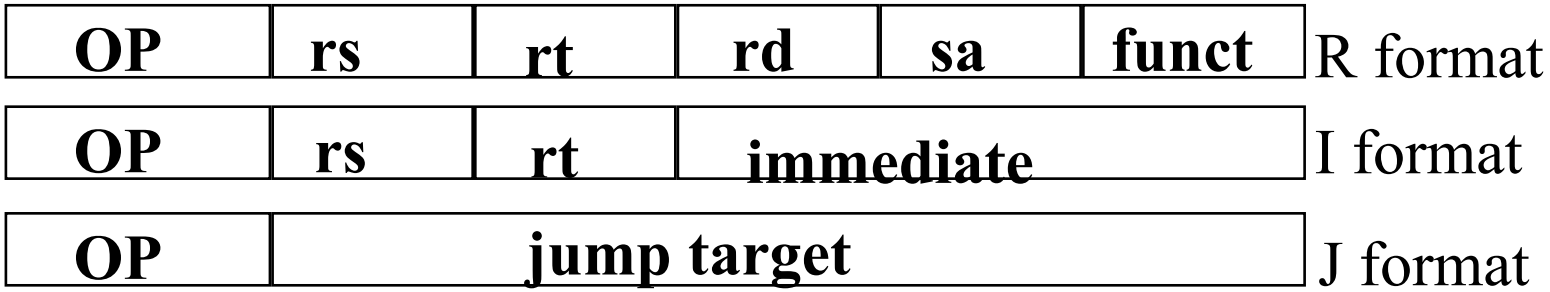
MIPS R3000 Instruction Set Architecture (ISA)

Instruction Categories

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

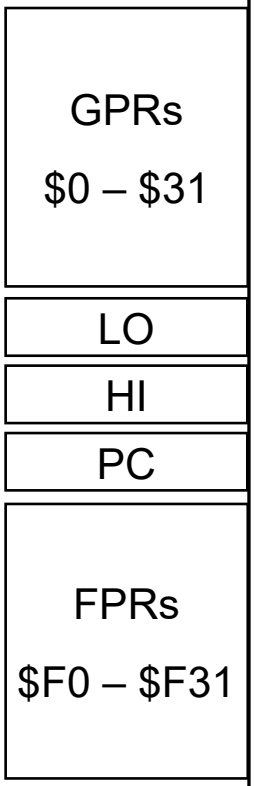


3 Instruction Formats:



Overview of the MIPS Registers

- **32 General Purpose Registers (GPRs)**
  - **32-bit registers are used in MIPS32**
  - **Register 0 is always zero**
  - **Any value written to R0 is discarded**
- **Special-purpose registers LO and HI**
  - **Hold results of integer multiply and divide**
- **Special-purpose program counter PC**
- **32 Floating Point Registers (FPRs)**
  - **Floating Point registers can be either 32-bit or 64-bit**
  - **A pair of registers is used for double-precision floating-point**



MIPS Register Convention

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
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

RISC - Reduced Instruction Set Computer

RISC philosophy

- **fixed instruction lengths**
- **load-store instruction sets**
- **limited addressing modes**
- **limited operations**

MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, HP (Compaq) Alpha, …

Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

MIPS Arithmetic

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)

- Example

C code: `A = B + C`

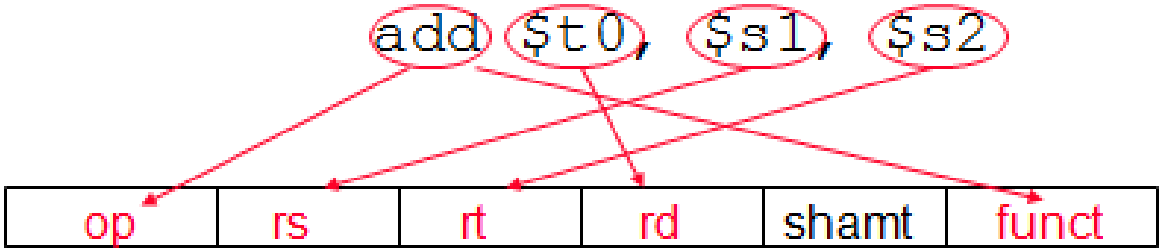
MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

- Using the natural number of operands for an operation (e.g. addition) conforms to the design principle of keeping the hardware simple.

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):



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

Temporary Variables

- **Regularity of instruction format requires that expressions get mapped to a sequence of binary operations with temporary results being stored in temporary variables.**

- **Example**

C code: `f = (g + h) - (i + j);`

Assume f, g, h, i, j are in $s0 through $s4 respectively

MIPS code:

```
add $t0, $s1, $s2 # $t0 = g+h
add $t1, $s3, $s4 # $t1 = i+j
sub $s0, $t0, $t1 # f = $t0 - $t1
```

Compiling a C Assignment Using Registers

EXAMPLE

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

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

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS code?

ANSWER

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, `$t0` and `$t1`, which correspond to the temporary variables above:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Registers vs. Memory

- Operands for arithmetic instructions must be registers,
  — only 32 integer registers are available

- Compiler associates variables with registers

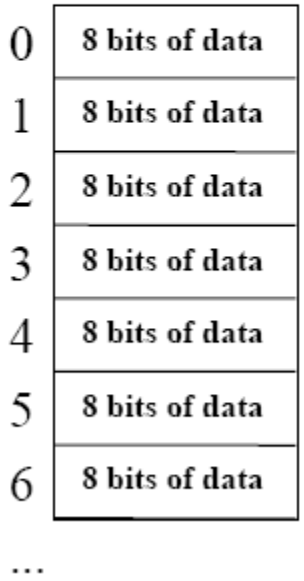
- When too many variable are used to fit in 32 registers, the compiler must allocate temporary space in memory and then load and store temporary results to/from memory.
- The compiler tries to put most frequently occurring variables in registers.
- The extra temporary variables must be “spilled” to memory.

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

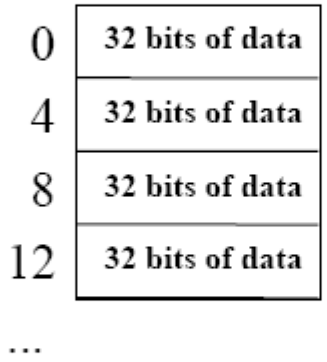
Memory Organization

- Viewed as a large, single-dimension array, where a memory address is an index into the array
- MIPS systems address memory in byte chunks
- The memory address (= index) points to a byte in memory.
- This is called "Byte addressing"



Memory Organization

- Most data items are grouped into words
- A MIPS word is 4 bytes or 32 bits



Registers also hold 32 bits of data

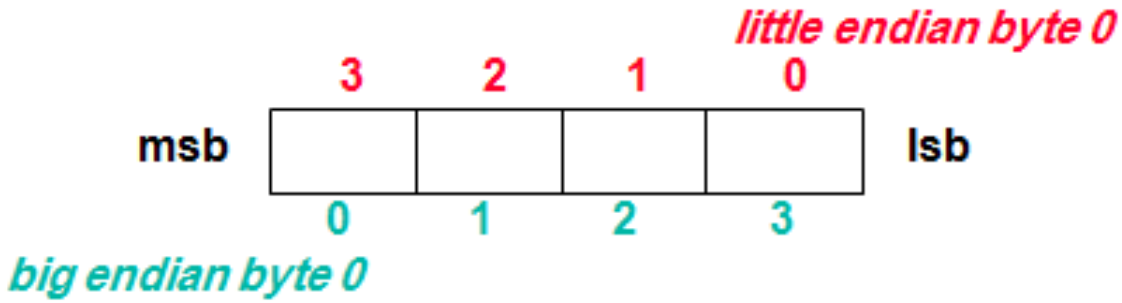
- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned (alignment restriction)
- Bytes can be accessed from left to right (big endian) or right to left (little endian).

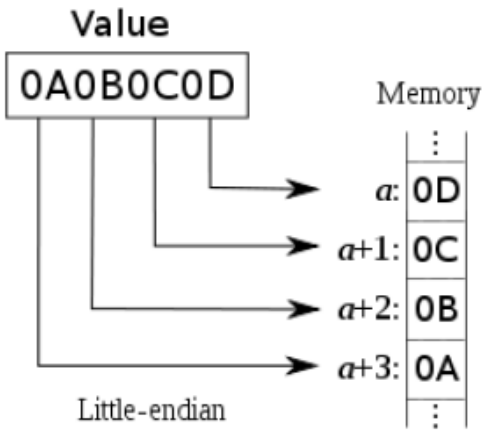
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- Big Endian: leftmost byte is word address

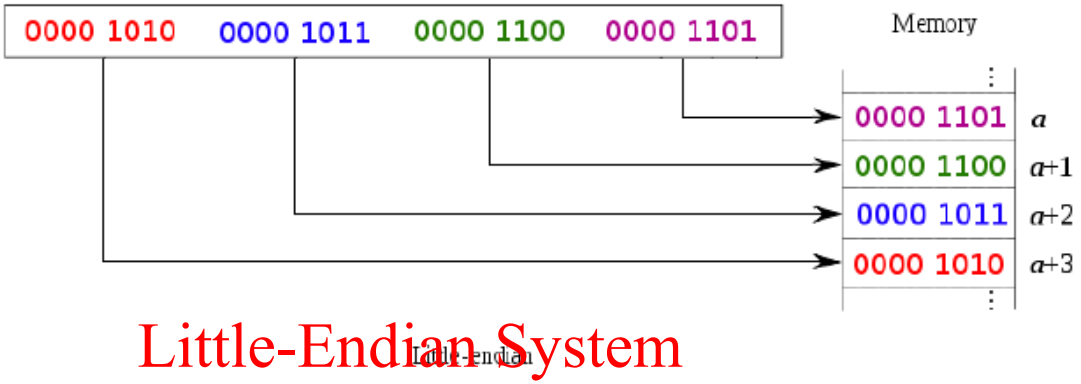
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

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



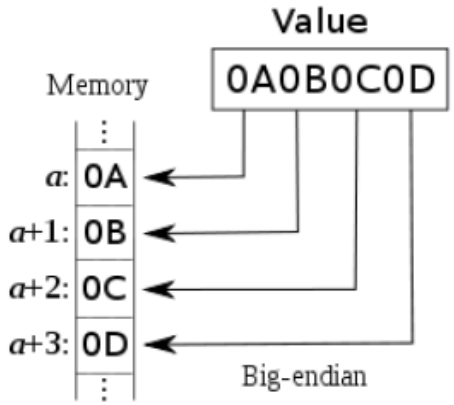


Hexadecimal Representation

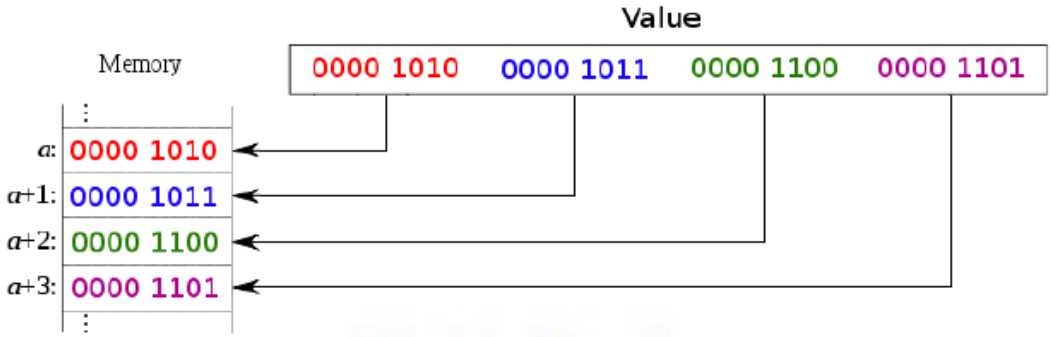


Little-Endian System

Binary Representation



Hexadecimal Representation

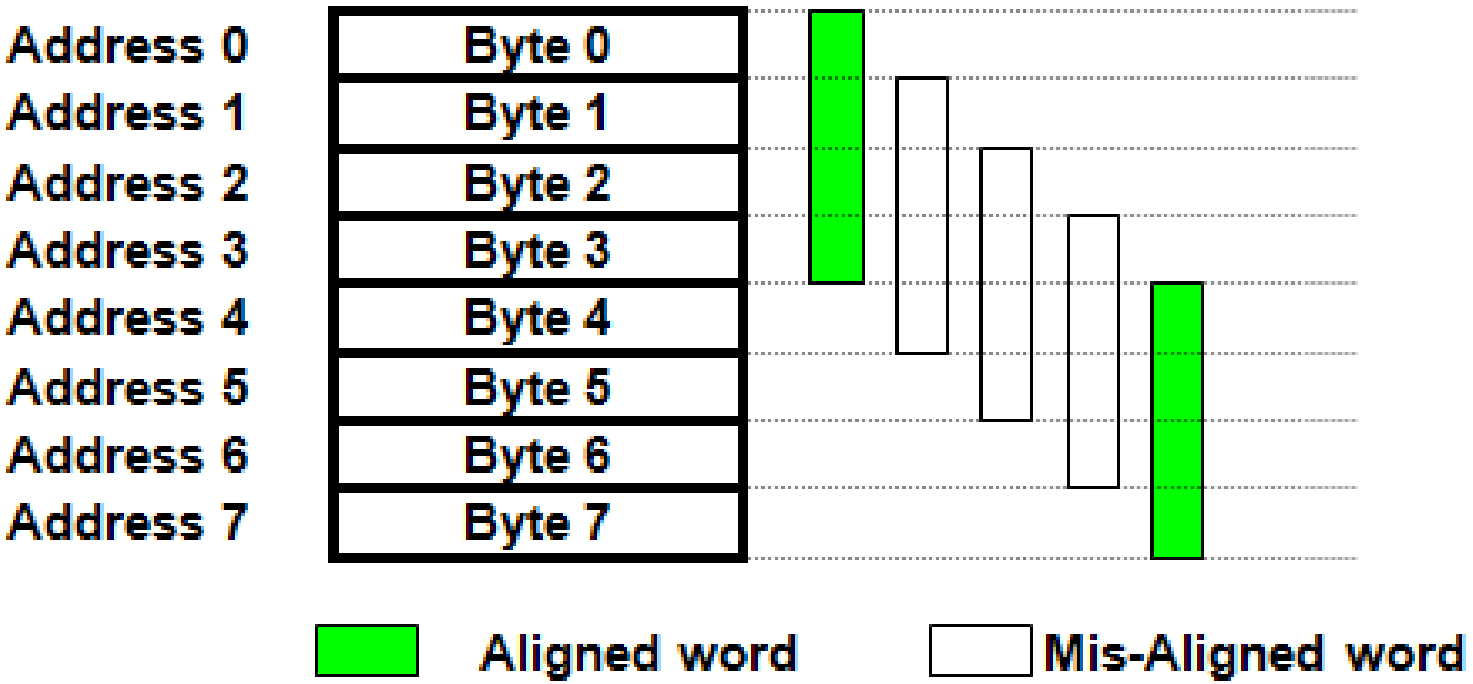


Big-Endian System

Binary Representation

Word alignment: Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word

Example: If a word consists of 4 bytes, then:



MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

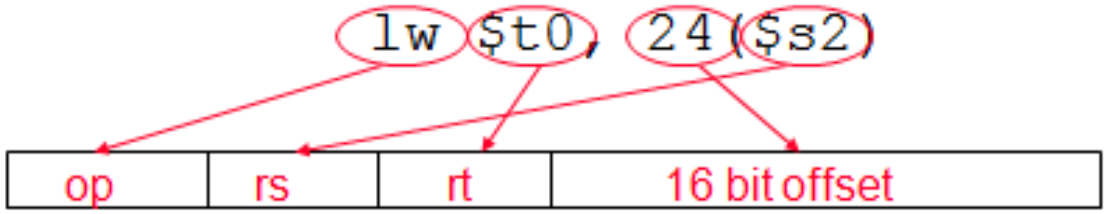
```
lw   $t0, 4($s3)   #load word from memory

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

- The data is 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 to the offset value
  - A 16-bit field meaning 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
  - Note that the offset can be positive or negative

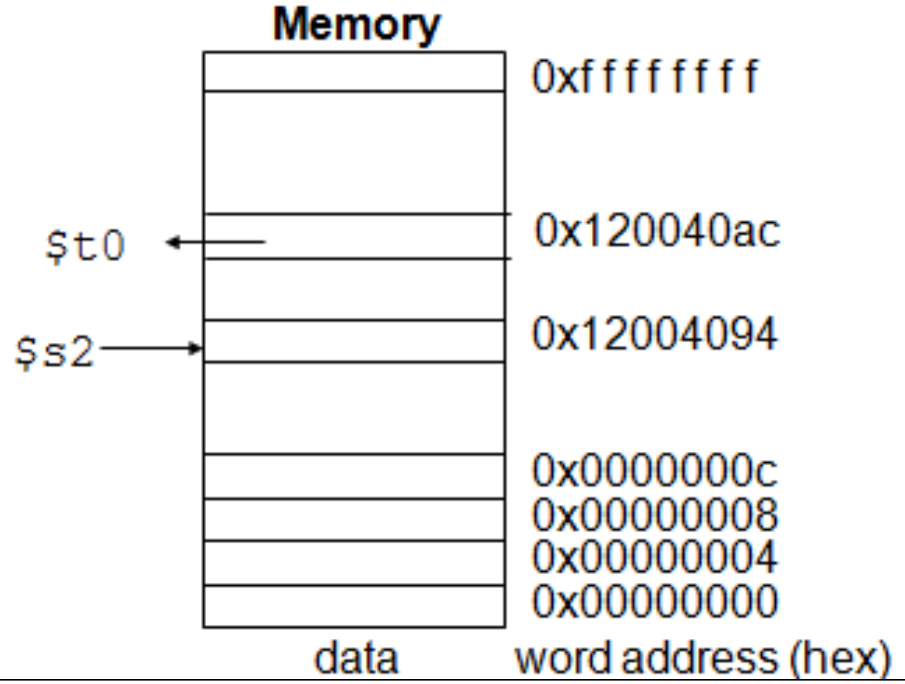
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s2 =

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



Load and Store

- All arithmetic instructions operate on registers
- Memory is accessed through load and store instructions

- An example C code: `A[12] = h + A[8];`

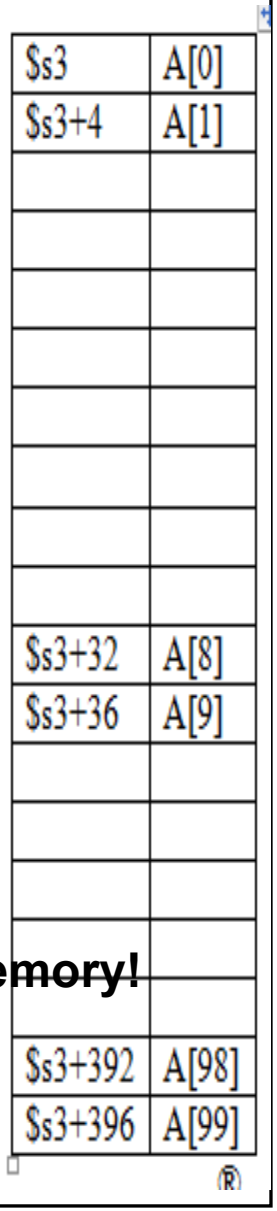
  Assume that $s3 contains the base address of A

  MIPS code:

```
lw $t0, 32($s3)
add $t0, $t0, $s2
sw $t0, 48($s3)
```

- Note: sw (store word instruction) has destination last.
- Note: remember arithmetic operands are registers, not memory!

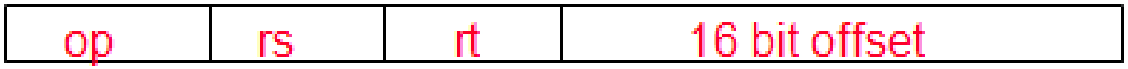
  This is invalid: `add 48($s3), $s2, 32($s3)`



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?

Memory Operand Example 1

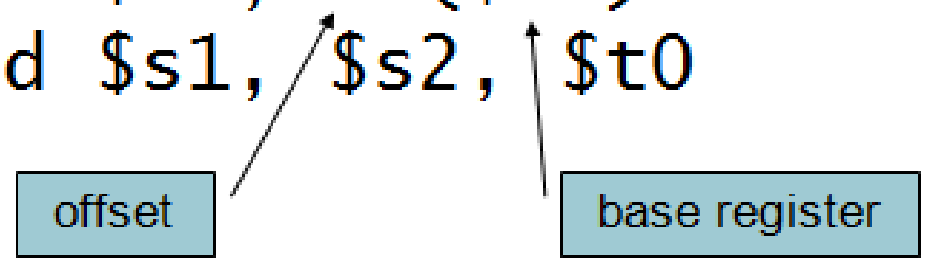
- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```



Compiling Using Load and Store

EXAMPLE

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

ANSWER

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select A[8], and the add instruction places the sum in $t0:

```
lw   $t0,32($s3)  # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 (4 × 12) as the offset and register $s3 as the base register.

```
sw   $t0,48($s3)  # Stores h + A[8] back into A[12]
```

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

MIPS register 0 ($zero) is the constant 0

Cannot be overwritten

Useful for common operations

E.g., move between registers

```
add $t2, $s1, $zero
```

Example of addi

addi instruction is represented in I format
rt for destination
rs for source

addi $t0,$t1,10

Now, we can construct the full instruction in binary:

Opcode	Source Register (rs)	Destination Register (rt)	Immediate
001000	01001	01000	0000000000001010

The Constant Zero

- **MIPS register 0 ($zero) is the constant 0**
  - **Cannot be overwritten**
- **Useful for common operations**
  - **E.g., move between registers**

```
add $t2, $s1, $zero
```

Logical Operations

- **Instructions for bitwise manipulation**

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

- **shamt: how many positions to shift**
- **Shift left logical**
  - **Shift left and fill with 0 bits**
  - **`sll` by *i* bits multiplies by 2^i**
- **Shift right logical**
  - **Shift right and fill with 0 bits**
  - **`srl` by *i* bits divides by 2^i (unsigned only)**

Note: The source register (rs) is not used in shift instructions, so it's set to 0.

Example

```
sll  $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits
```

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

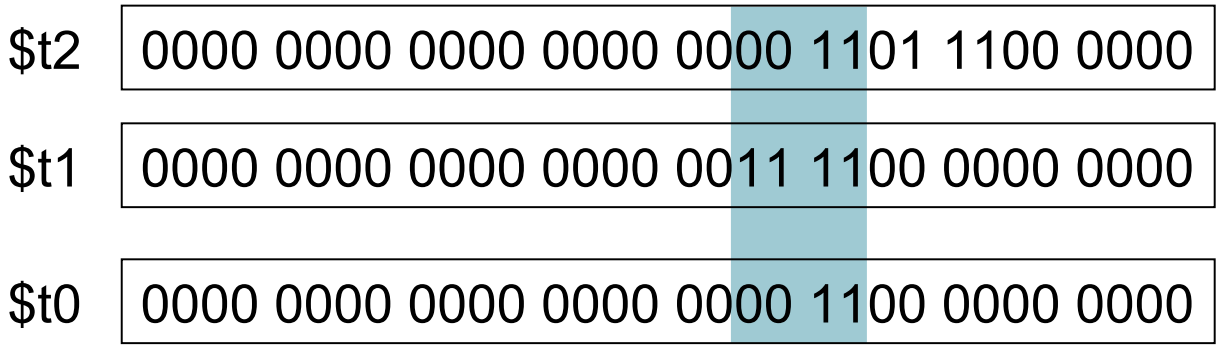
op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of `sll` is 0 in both the op and funct fields, rd contains 10 (register `$t2`), rt contains 16 (register `$s0`), and shamt contains 4. The rs field is unused and thus is set to 0.

AND Operations

- **Useful to mask bits in a word**
  - **Select some bits, clear others to 0**

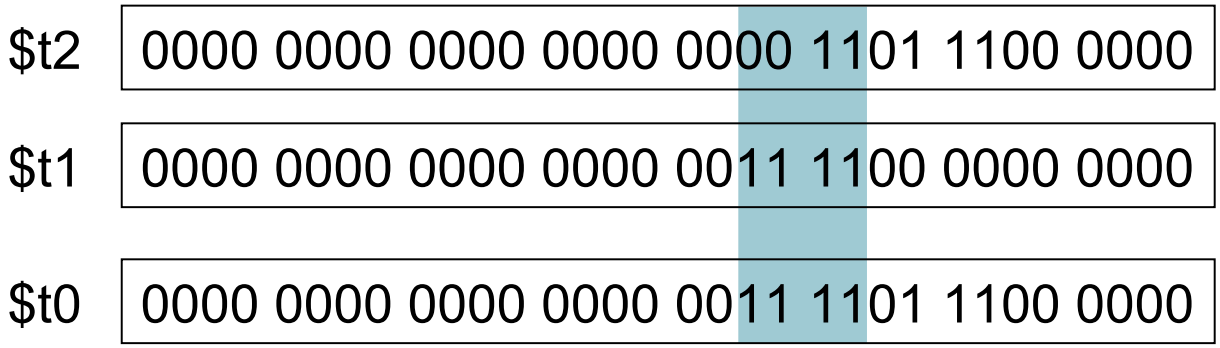
```
and $t0, $t1, $t2
```



OR Operations

- **Useful to include bits in a word**
  - **Set some bits to 1, leave others unchanged**

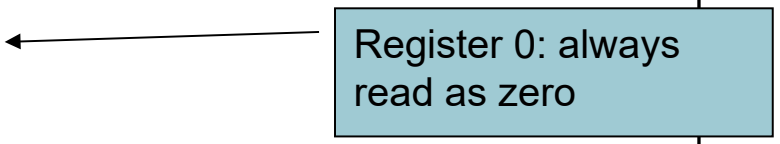
```
or $t0, $t1, $t2
```



NOT Operations

- **Useful to invert bits in a word**
  - **Change 0 to 1, and 1 to 0**
- **MIPS has NOR 3-operand instruction**
  - **a NOR b == NOT (a OR b)**

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



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

So far we've learned:

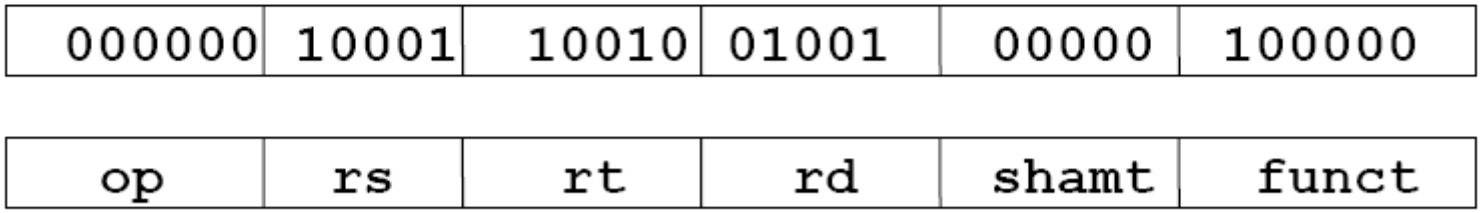
- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **Instruction** **Meaning**

```
add $s1, $s2, $s3       $s1 = $s2 + $s3
sub $s1, $s2, $s3       $s1 = $s2 - $s3
lw $s1, 100($s2)        $s1 = Memory[$s2+100]
sw $s1, 100($s2)        Memory[$s2+100] = $s1
```

Machine Language

- Instructions, registers and data words are 32 bit long
  - Example: `add $t1, $s1, $s2`
  - Registers have numbers/indeces: `$t1=9, $s1=17, $s2=18`
- Instruction Format:



- Guess what do the field names stand for?

Translating a MIPS Assembly Instruction into a Machine Instruction

EXAMPLE

Let's do the next step in the refinement of the MIPS language as an example. We'll show the real MIPS language version of the instruction represented symbolically as

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

first as a combination of decimal numbers and then of binary numbers.

ANSWER

The decimal representation is

0	17	18	8	0	32

Each of these segments of an instruction is called a *field*. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation (17 = `$s1`), and the third field gives the other source operand for the addition (18 = `$s2`). The fourth field contains the number of the register that is to receive the sum (8 = `$t0`). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register `$s1` to register `$s2` and places the sum in register `$t0`.

This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-format type for data transfer instructions
  - The other format was R-type for register (add and sub)
- Example: `lw $t0, 32($s2)`

35	18	8	32

op	rs	rt	16 bit number

- Where's the compromise?

MIPS instruction encoding.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

EXAMPLE

Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw   $t0,1200($t1)# Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1)# Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

ANSWER

For convenience, let's first represent the machine language instructions using decimal numbers. From Figure 2.5, we can determine the three machine language instructions:

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

The `lw` instruction is identified by 35 (see Figure 2.5) in the first field (op). The base register 9 (`$t1`) is specified in the second field (rs), and the destination register 8 (`$t0`) is specified in the third field (rt). The offset to select `A[300]` ($1200 = 300 \times 4$) is found in the final field (address).

The `add` instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to `$s2`, `$t0`, and `$t0`.

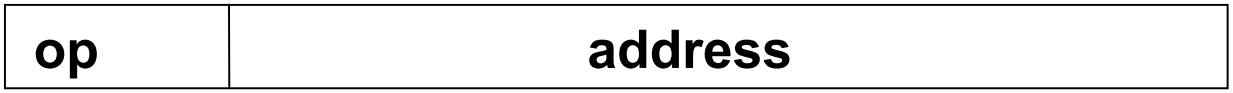
The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

Since $1200_{ten} = 0000\ 0100\ 1011\ 0000_{two}$, the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Addressing in Jumps

- **J format format (jump format – j, jal)**

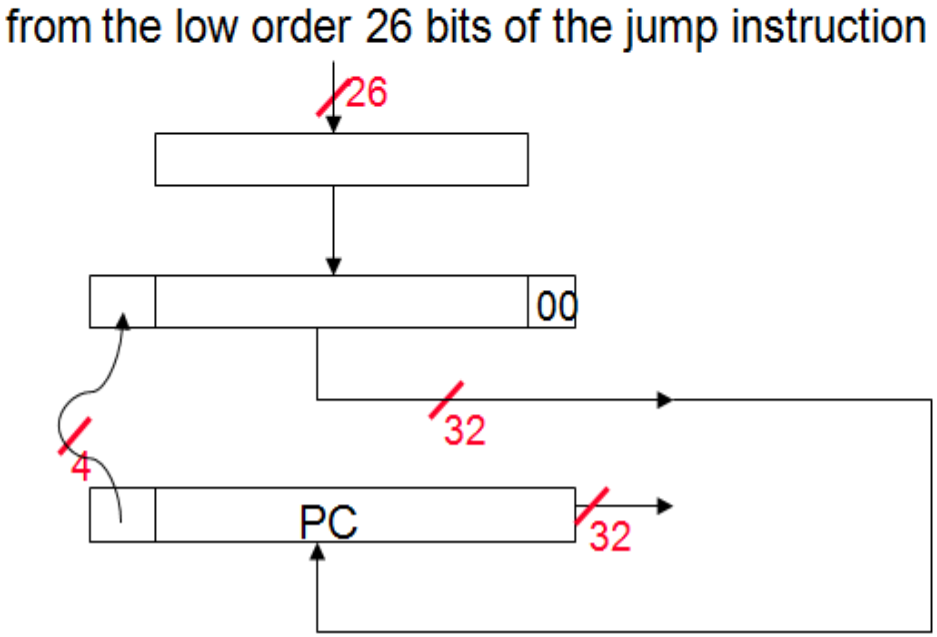


6-bits 26-bits

- **Example: j 10000**

6-bits 26-bits

- Instruction Format (J Format):



Target Addressing Example

- **Loop code example**
  - **Assume Loop at location 80000**

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

Design Principles

- **Simplicity favors regularity**
  - **All instructions 32 bits**
  - **All instructions have 3 operands**
- **Smaller is faster**
  - **Only 32 registers**
- **Good design demands good compromises**
  - **All instructions are the same length**
  - **Limited number of instruction formats: R, I, J**
- **Make common cases fast**
  - **16-bit immediate constant**
  - **Only two branch instructions**