COMPUTER ORGANIZATION AND DESIGN

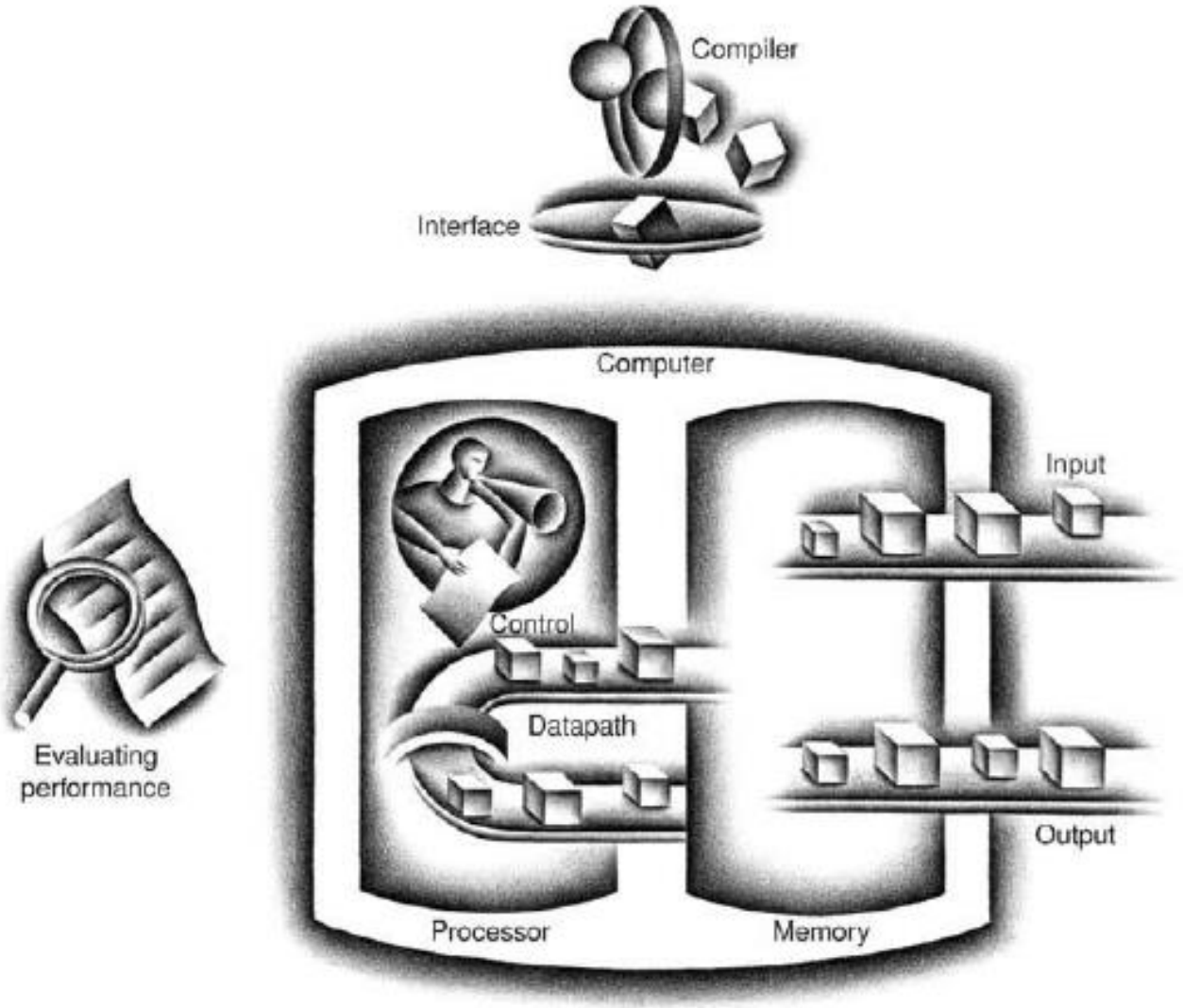
The Hardware/Software Interface

Chapter 2

Instructions:

Language of the Computer

The Classic Components of a Computer



What Is Computer Architecture?

- Computer Architecture
  - Instruction Set Architecture + Computer Organization
- Instruction Set Architecture (ISA)
  - WHAT the computer does (logical view)
- Computer Organization
  - HOW the ISA is implemented (physical view)

Instruction Set

- **Instruction set:** The vocabulary of commands understood by a given architecture.
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Instruction Set

User Level

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Interfacing

Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Assembler

Hardware Level

Binary machine language program (for MIPS)

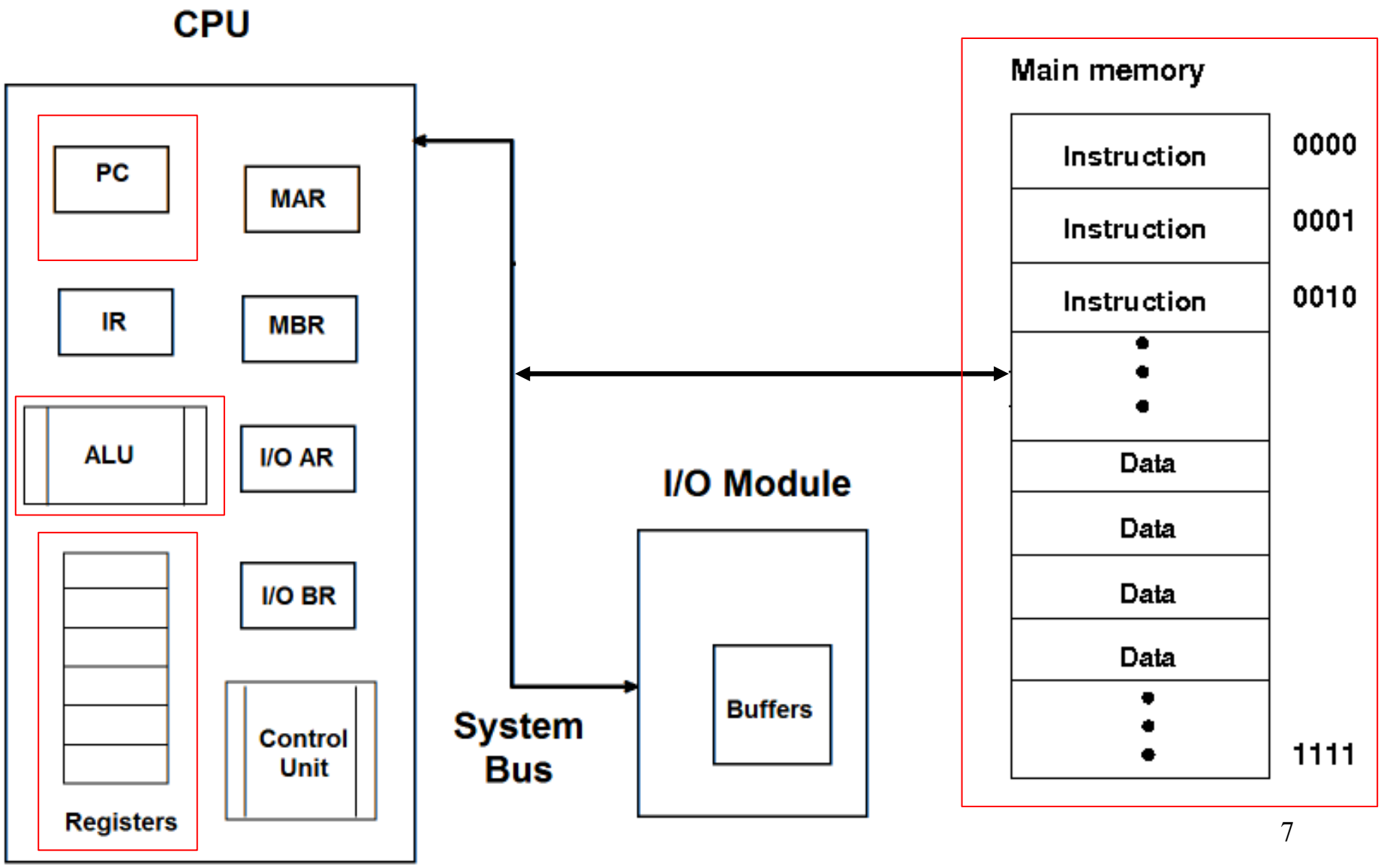
```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Stored-program
The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored-program computer.

The MIPS Instruction Set

- Used as the example throughout the course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Computer Components: Top Level View

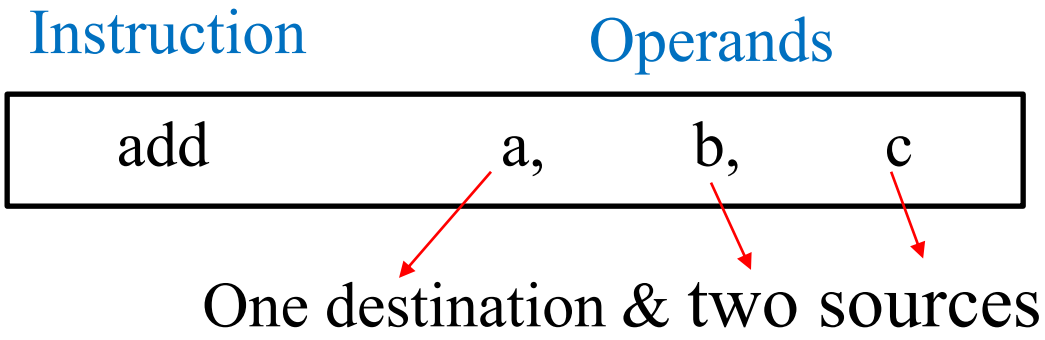


Arithmetic Operations

- Every computer must be able to perform arithmetic and logical operations.

- Arithmetic operations: add, subtract, multiply & divide are the building blocks for any arithmetic operation.

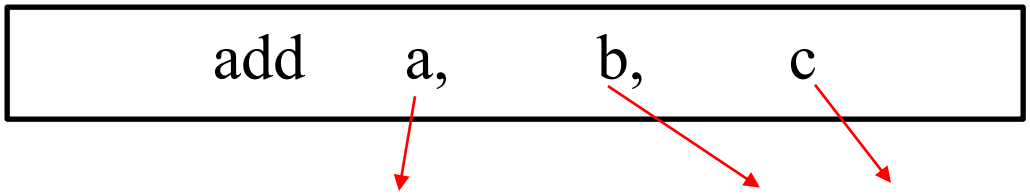
- The MIPS assembly language: (a, b, and c are variables)



- All arithmetic operations have this form.

Arithmetic Operations

- The MIPS assembly language: (**a**, **b**, and **c** are variables)

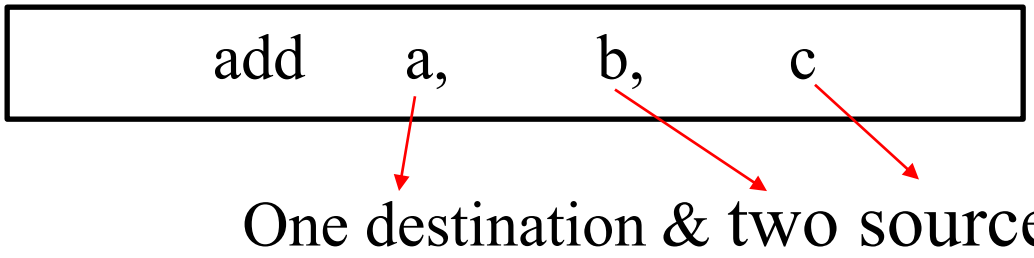


- **Example:** place the sum of variables **b**, **c**, **d**, and **e** into variable **a.**
  - Computer performs this arithmetic operation as a set of add instructions sequentially.

```
add a, b, c      # The sum of b and c is placed in a.
add a, a, d      # The sum of b, c, and d is now in a.
add a, a, e      # The sum of b, c, d, and e is now in a.
```

Arithmetic Operations

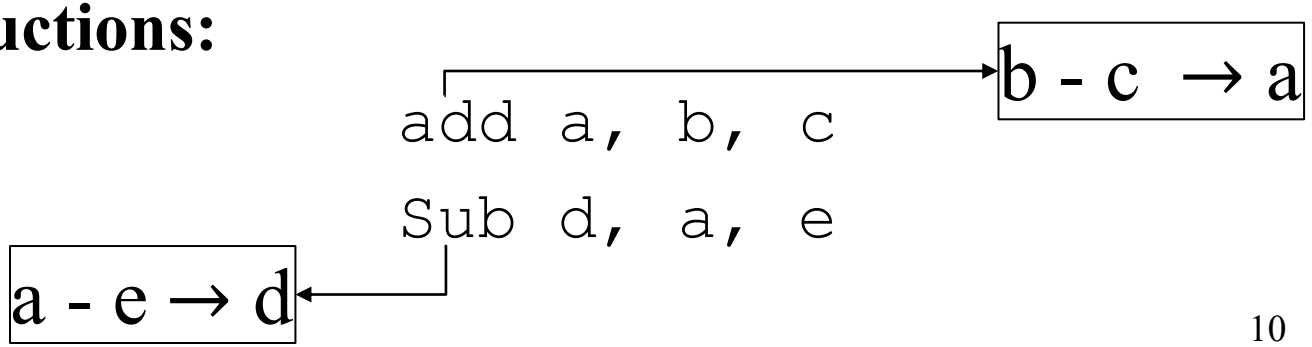
- The MIPS assembly language: (**a**, **b**, and **c** are variables)



- Example: Compiling Two C Assignment Statements into MIPS

```
a = b + c;
d = a - e;
```

MIPS instructions:



Arithmetic Operations

- **Example:** Compiling a Complex C Assignment into MIPS

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

MIPS instructions: a set of instructions in a sequential order

```
add t0,g,h # temporary variable t0 contains g + h
```

```
add t1,i,j  # temporary variable t1 contains i + j
```

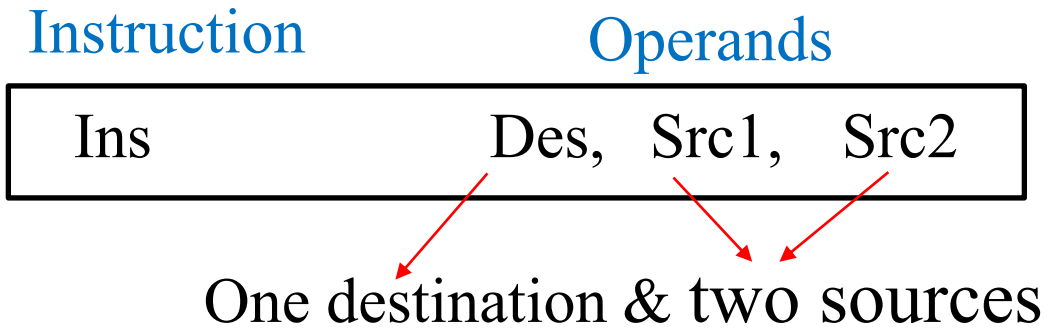
```
sub f,t0,t1 # f gets t0 - t1, which is (g + h)-(i + j)
```

Design Principles

Simplicity favors regularity

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

Operands



In general, operands can be:

- Register Operands
- Memory Operands
- Immediate Operands
- Constant Zero

Register Operands

- Arithmetic instructions use register operands
- MIPS has a **32 × 32-bit register file**
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- **The word** is the natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

Register Operands

- Resister names (or numbers):

Name	Register	Usage	
$zero	$0	Always 0	(forced by hardware)
$at	$1	Reserved for assembler use	
$v0 – $v1	$2 – $3	Result values of a function	
$a0 – $a3	$4 – $7	Arguments of a function	
$t0 – $t7	$8 – $15	Temporary Values	
$s0 – $s7	$16 – $23	Saved registers	(preserved across call)
$t8 – $t9	$24 – $25	More temporaries	
$k0 – $k1	$26 – $27	Reserved for OS kernel	
$gp	$28	Global pointer	(points to global data)
$sp	$29	Stack pointer	(points to top of stack)
$fp	$30	Frame pointer	(points to stack frame)
$ra	$31	Return address	(used by jal for function call)

Design Principles

Smaller is faster

- A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

- Main memory has millions of locations

Register Operand Example

- C code:

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

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

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

Register Operand Example

- Compiling Two C Assignment Statements into MIPS

```
a = b + c;
d = a – e;
```

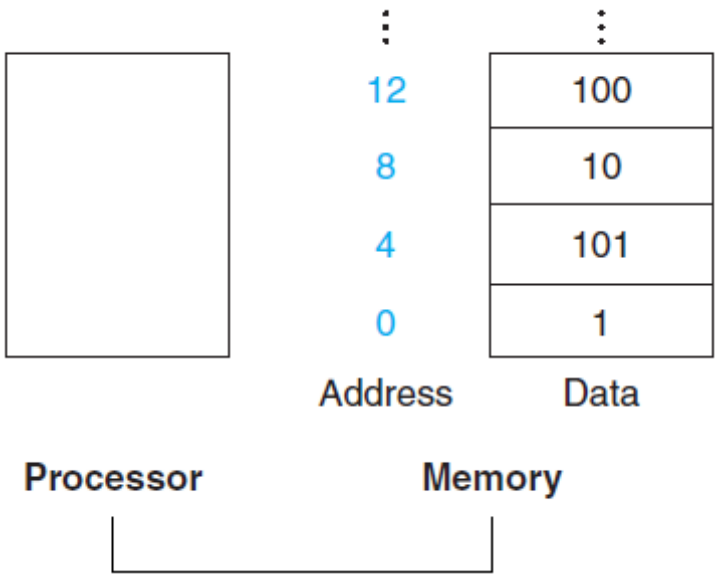
a, b, c, d and e in registers $s0, $s1, $s2, $s3 and $s4

- Compiled MIPS code:

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

Memory Operands

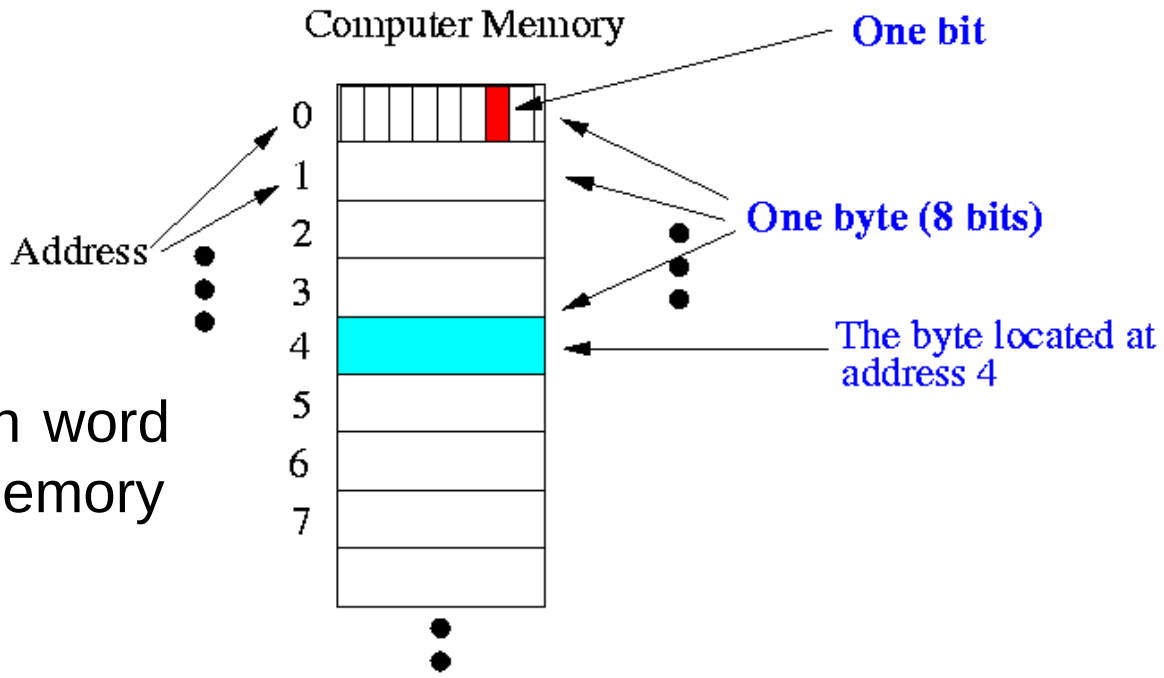
- Main memory used for composite data
  - Arrays, structures, dynamic data



- To apply arithmetic operations
  - **Load values** from memory into registers
  - **Store result** from register to memory

Memory Operands

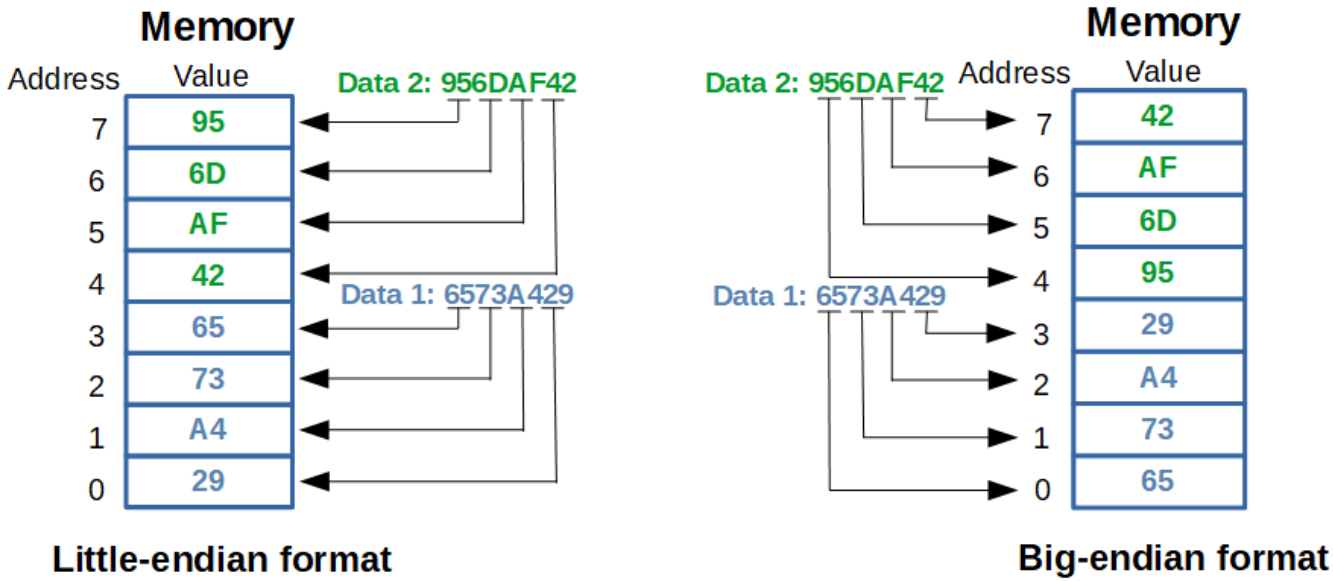
- Memory is byte addressed
  - Each address identifies an 8-bit (1 byte)
- Words are aligned in memory
  - Address must be a multiple of 4 (i.e., 4 locations)



Word size is 32. Thus, each word will take 4 locations in the memory

Memory Operands

- MIPS is Big Endian machine
  - Most-significant byte at least address of a word
  - Little Endian: least-significant byte at least address



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
```

Offset = 8*4=32

base register consists of the address of the first element in A

Memory Operand Example 2

- C code:

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

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

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible

Immediate Operands

- Constant data specified in an instruction

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

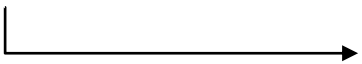
- No subtract immediate instruction
  - Just use a negative constant

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

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
```



- The **move instruction** (pseudo-instruction):
  - move between registers

```
move $t2, $s1
```

Design Principles

Make the common case fast

- Small constants are common
- Immediate operand avoids a load instruction

MIPS Instructions

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, . . . , $t0, $t1, . . .`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three operands; data in registers
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Data from register to memory

MARS simulator

- **MARS** (MIPS Assembler and Runtime Simulator): An IDE for MIPS Assembly Language Programming

- **MARS** is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's Computer Organization and Design.

- **Features:**
  - GUI with point-and-click control and integrated editor
  - Easily editable register and memory values, similar to a spreadsheet
  - Display values in hexadecimal or decimal
  - And many others

MARS simulator

- Download the MARS simulator. On the MARS download page, select the "Download MARS" button.
- For detailed directions, refer to the MARS tutorial , this website or watch this video on YouTube
- Here are a few example MIPS assembly programs you may download and try:
  area_triangle.asm, arith_exmpl.asm, sum_1to100.asm.

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

MIPS Instructions

add unsigned	`addu $1,$2,$3`	$1=$2+$3	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	`subu $1,$2,$3`	$1=$2-$3	Values are treated as unsigned integers, not two's complement integers
add immediate unsigned	`addiu $1,$2,100`	$1=$2+100	Values are treated as unsigned integers, not two's complement integers

add	`add $1,$2,$3`	$1=$2+$3
subtract	`sub $1,$2,$3`	$1=$2-$3
add immediate	`addi $1,$2,100`	$1=$2+100

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

$0000\ 0000\ 0000\ 00000\ 000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$

Shift left operation

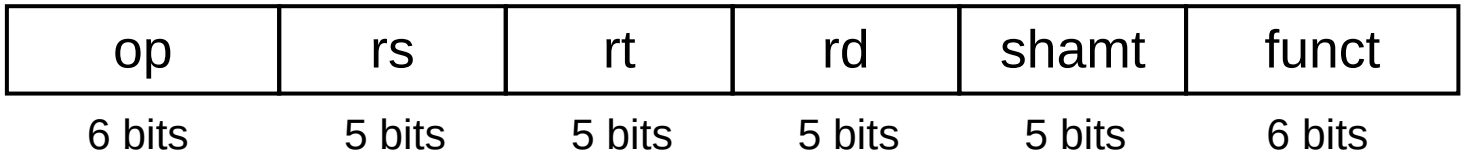
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$

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

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

R format instruction

Shift Operations

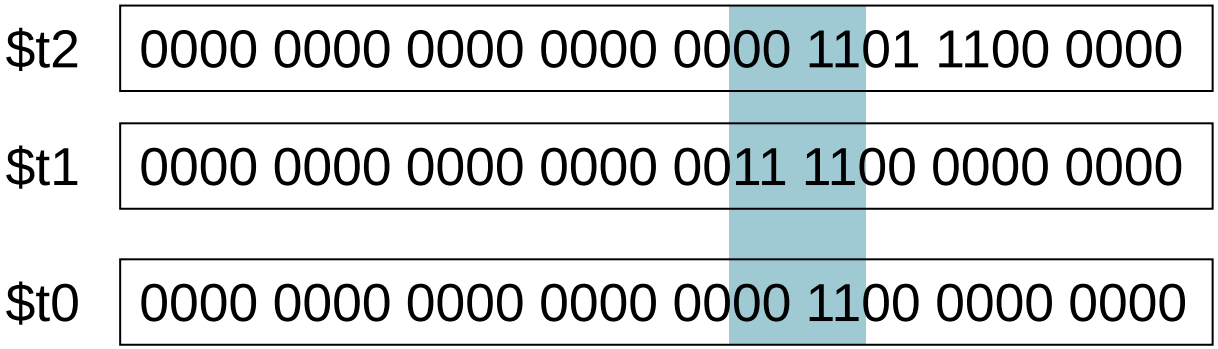


- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

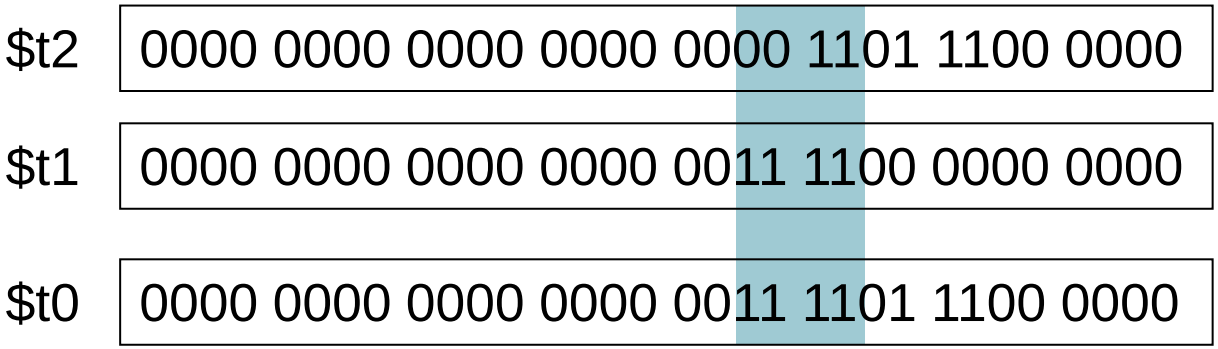
```
and $t0, $t1, $t2
```



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
```

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 Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory

Conditional Operations

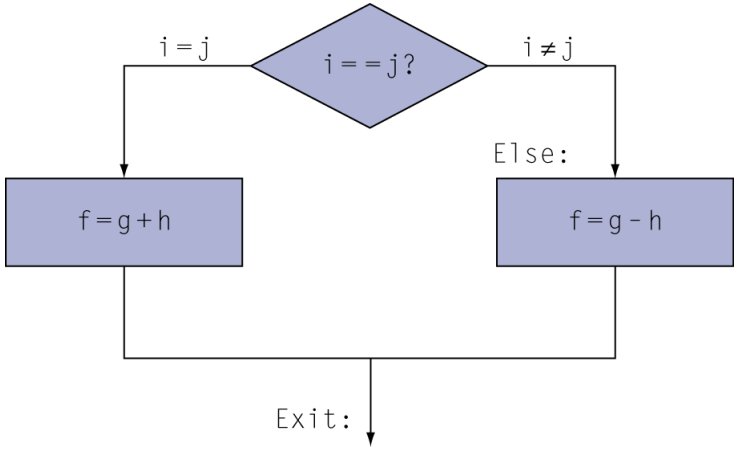
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled `L1`;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled `L1`;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling If Statements

C code

```
temp = a;
if ( a != b )
temp =(a+b)/2;
temp += c;
```

MIPS Instructions

```
        move $t0, $s0
        beq $s0, $s1, ENDIF
        add $t0, $t0, $s1
        srl $t0, $t0, 1
ENDIF:  add $t0, $t0, $s2
```

Assume that `$s0` = a, `$s1` = b, `$s2` = c, `$t0` = temp.

Compiling Loop Statements

- C code:

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

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

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

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Example 1

- The following C code:

```
int i;
for ( i = 0; i < numVals; i++ )
{    int x = A[i];
     x *= 2;
     A[i] = x;
}
```

$s6 = A ; $s7 = numVals ;

$t0 = I ; $t4 = x

Example 1

- MIPS Instructions:

```
            addi $t0, $zero, 0
LOOP:       slt $t1, $t0, $s7
            beq $t1, $zero, ENDLOOP
            sll $t1, $t0, 2
            add $t1, $s7, $t1
            lw $t4, 0($t1)
            sll $t4, $t4, 1
            sw $t4, 0($t1)
            addi $t0, $t0, 1
            j LOOP
ENDLOOP:
            ⋮
```

Example 2

- The following C code:

```
sum = 0;
for ( i = 0; i < n; i++ )
{
    sum += A[i];
}
```

Assume that `$s0` = i, `$s1` = n, `$s2` = A, `$s3` = sum, and that A is an array of `int`.

Example 2

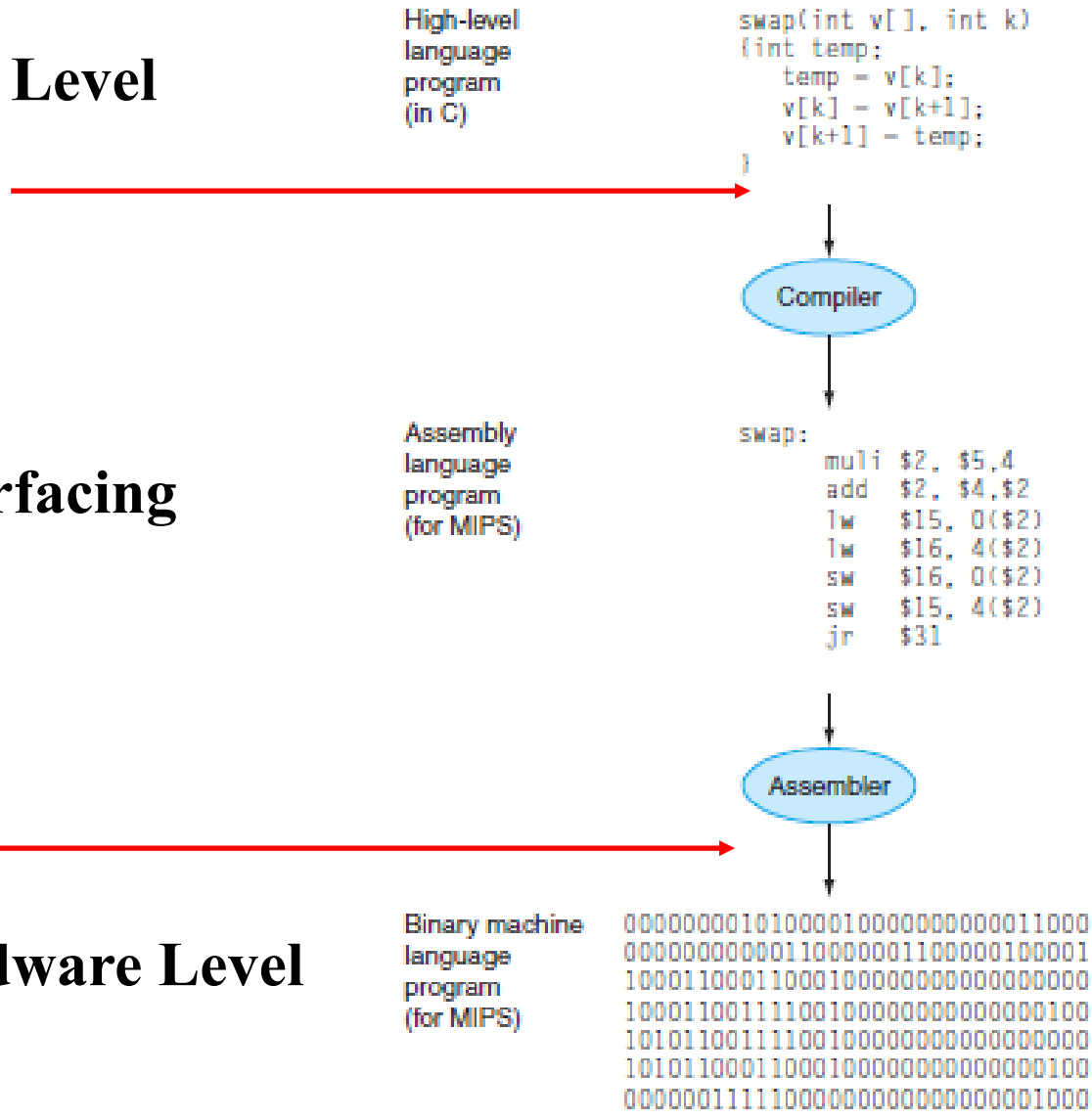
- MIPS Instructions:

```
        move $s3, $zero          # sum = 0
        move $s0, $zero          # i = 0
LOOP:   slt $t0, $s0, $s1        # t0 is 1 if i < n; 0 if i >= n
        beq $t0, $zero, END      # goto END if i >= n
        sll $t0, $s0, 2          # t0 = i * 4
        add $t0, $t0, $s2        # t0 is address of A[i]
        lw $t1, 0 ($t0)          # t1 = A[i]
        add $s3, $s3, $t1        # sum += A[i]
        addi $s0, $s0, 1         # i += 1 (or i++)
        j LOOP
END:    ...
```

MIPS Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address

Representing Instructions



Representing Instructions

- The **binary digit (**called binary bit), One of the two numbers in base 2, 0 or 1, that are the components of information.

- **Machine language:** A binary representation used for communication within a computer system.

- **Instruction format:** A form of representation of an instruction composed of fields of binary numbers.

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

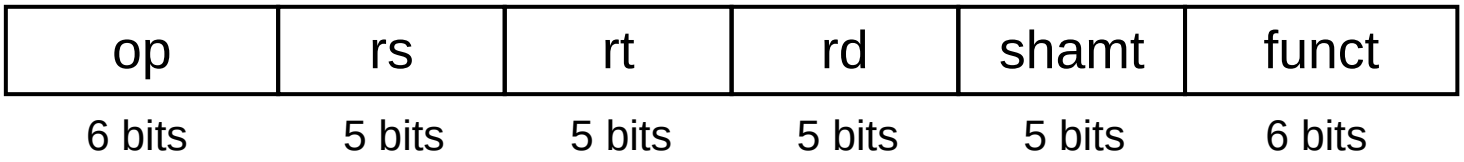
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS R-format Instructions



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

R-format Example

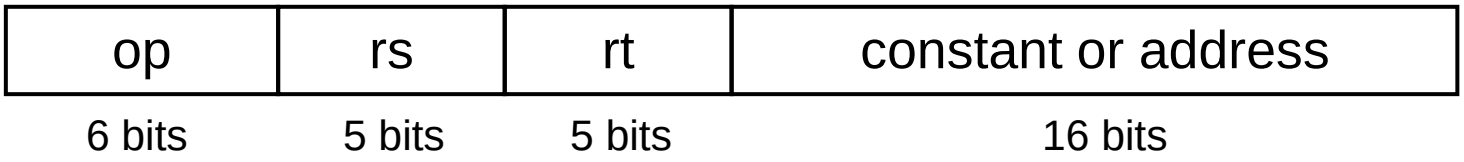
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

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

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

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

Representing Instructions (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];
```

1) MIPS Instructions:

```
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]
```

Representing Instructions (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];
```

2) MIPS machine code:

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

Representing Instructions (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:

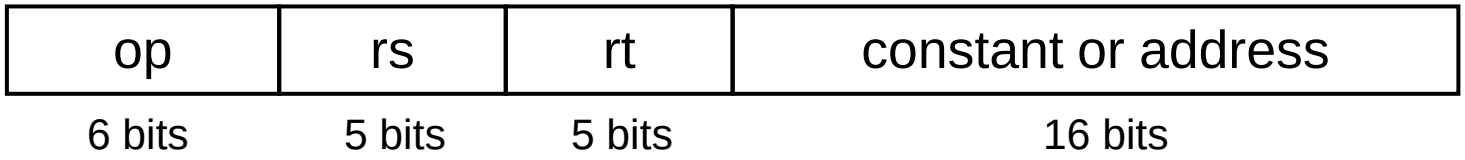
```
A[300] = h + A[300];
```

2) MIPS machine code:

op	rs	rt	rd	address/ shamt	funct
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Branch Addressing

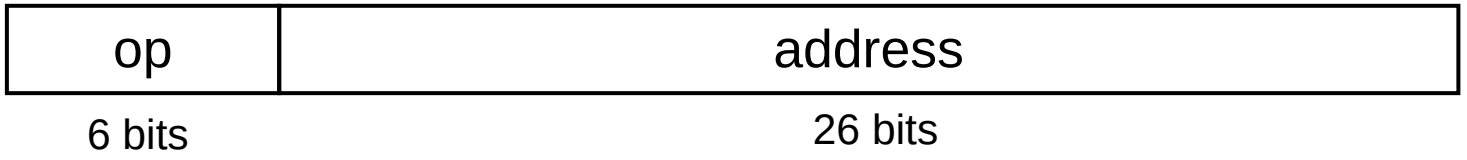
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



  - PC-relative addressing
    - Target address = PC + offset × 4
    - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

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

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

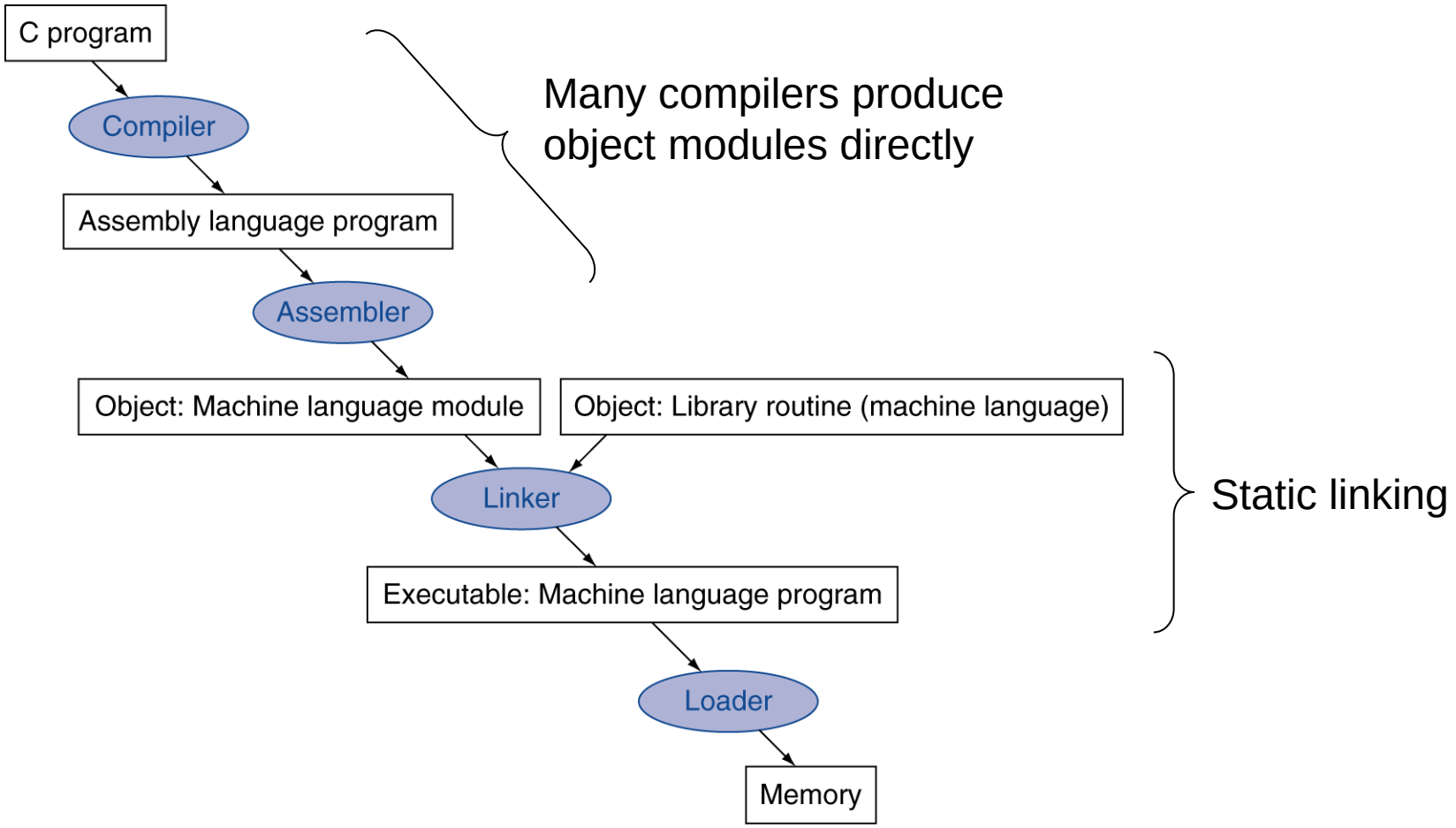
Design Principles

Good design demands good compromises

- Different formats complicate decoding, but allow 32-bit instructions uniformly

- Keep formats as similar as possible

Translation and Startup



Role of Assembler

- Convert pseudo-instructions into actual hardware instructions
  - pseudo-instrs make it easier to program in assembly
  - examples: “move”, “blt”, 32-bit immediate operands, etc.
- Convert assembly instrs into machine instrs
  - a separate object file (x.o) is created for each C file (x.c)
  - compute the actual values for instruction labels
  - maintain info on external references and debugging information

Assembler Pseudoinstructions

- Branch Pseudoinstructions (Branch if less than (blt))

The blt instruction compares 2 registers, treating them as signed integers, and takes a branch if one register is less than another.

```
blt $8, $9, 4
```

translates to

```
slt $1, $8, $9
bne $1, $0, 4
```

- Other Pseudoinstructions (Move (move))

The move pseudo instruction moves the contents of one register into another register.

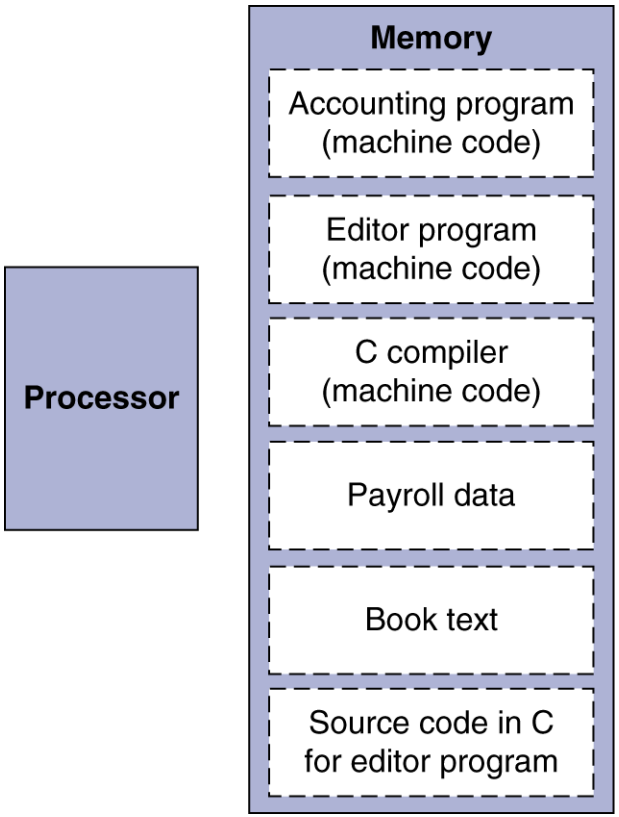
```
move $1, $2
```

translates to

```
add $1, $2, $0
```

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs