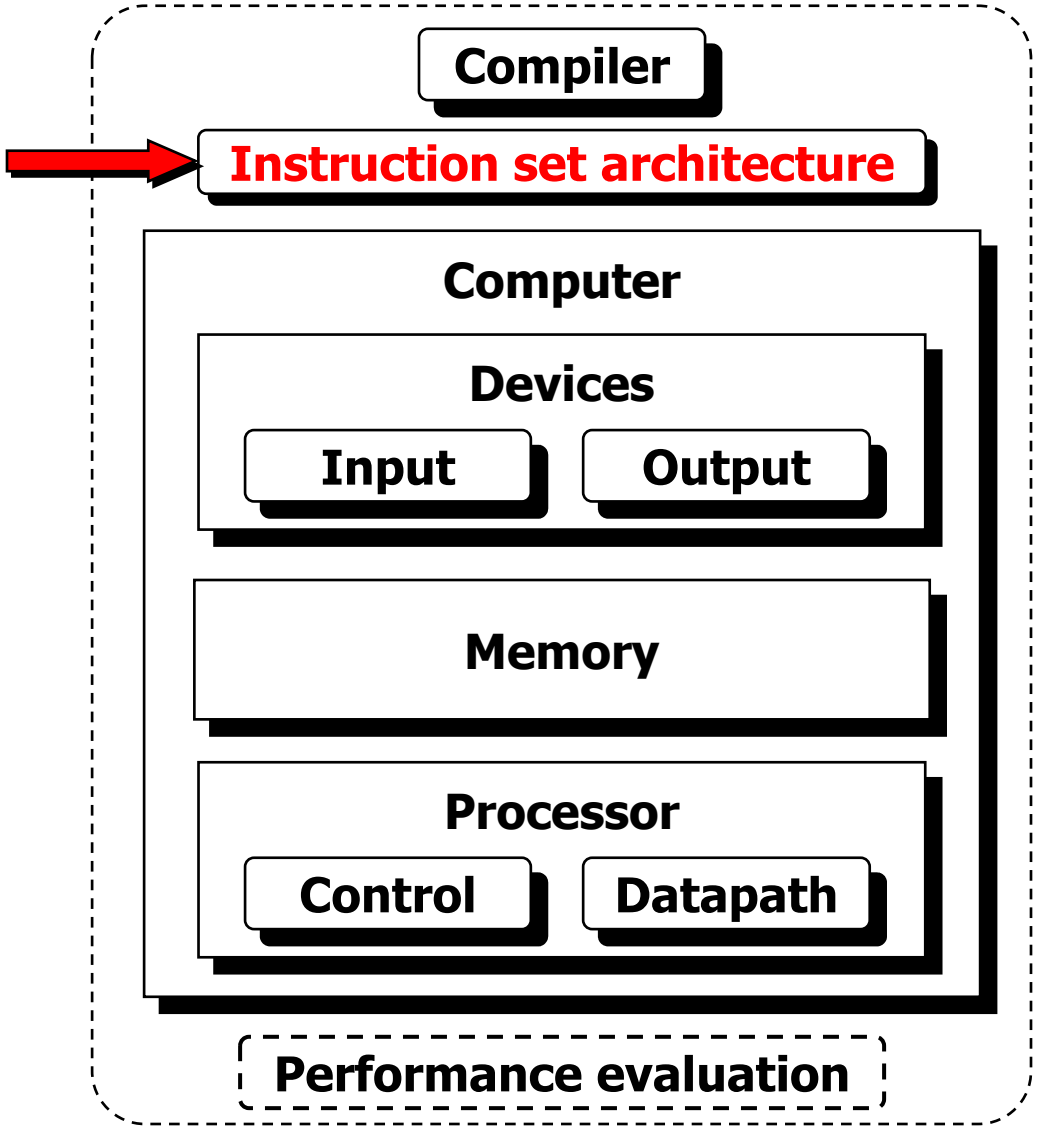
Instructions: Language of the Computer

Chapter 2



Agenda and Reading List

- Chapter goals
- Introduction (2.1, pp. 62-63)
- Below your program (1.3, pp. 13-14)
- From a high-level language to the language of hardware (1.3, pp. 14-16)
- Abstractions (1.4, p. 22)
- Instruction set architecture (ISA) (1.4, p. 22)
- The stored-program concept (2.1, p. 63)
- Operations (2.2, pp. 63-66)
- Operands (2.3, pp. 66-68)
- Registers (2.8, p. 105)
- Memory organization (2.3, pp. 68 and 70)
- Data transfer instructions (2.3, pp. 69 and 71)
- Small constant or immediate operands (2.3, pp. 72-73 and 2.10, pp. 111-113)

Agenda and Reading List (cont.)

- Sign and zero extension of immediates (2.4, p. 76)
- Logical (bitwise) instructions (2.6, pp. 87-89)
- 32-bit immediate operands (2.10, pp. 112-113)
- MIPS simulators
- Instruction formats (2.5, pp. 80-87 and 2.10, pp. 118-120)
- Program translation hierarchy (2.12, pp. 123-125)
- Decision making instructions (2.7, pp. 90-96)
- Addressing in branches and jumps (2.10, pp. 113-117)
- Supporting procedures (2.8, pp. 96-106)
- ~~Strings (2.9, pp. 106-111)~~
- CISC versus RISC architectures
- Instruction set styles
- Fallacies and pitfalls (2.19, pp. 159-161)
- Concluding remarks (2.29, pp. 161-163)

Chapter Goals

- to discuss a variety of relevant abstractions
- to introduce the stored-program concept
- to place the major components of software and hardware in perspective
- to present a subset of the MIPS assembly language
- to teach an instruction set that follows the "simplicity of the equipment" advice
- to show how an instruction set is represented in hardware
- to see the impact of programming languages and compiler optimization on performance
- to illustrate basic instruction set design principles
- to discuss the relationship between high-level languages and the more primitive one (i.e., how high-level constructs, such as if and loop statements, are expressed in assembly)
- to explain and contrast the different architecture alternatives

Introduction

- **Instructions:** words of a computer's language
- **Instruction set:** computer language vocabulary

- **Types of Instruction Sets:**
  - **CISC: Complex Instruction Set Computer**
    - Supplies a large number of complex instructions at the machine level resulting in shorter programs
    - **Examples:** Intel 80x86 and Motorola 680x0
  - **RISC: Reduced Instruction Set Computer**
    - Supplies a relatively limited number of simple instructions at the machine level resulting in faster execution
    - Almost all new instruction sets since 1982 followed the RISC style
    - **Examples:** MIPS, Sun Sparc, HP PA-RISC, PowerPC, DEC Alpha
    - Used by ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, TI, Toshiba, etc.
    - We will work with the **MIPS instruction set**

Introduction (cont.)

We will focus on architectural issues

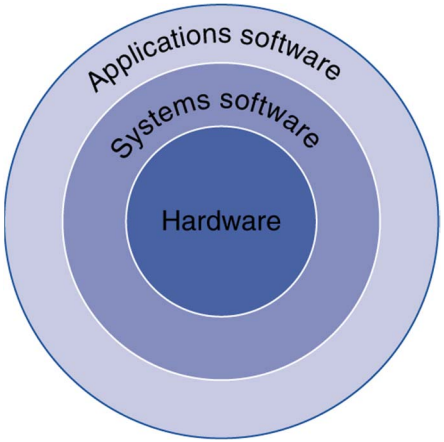
- basics of MIPS assembly language and machine code
- later, we will build a processor to execute these instructions

- **Computer designers' goal:** find an assembly language that makes it easy to build the hardware and the compiler while
  - maximizing performance
  - minimizing cost
  - minimizing power
  - reducing design time

- **Considerations in selecting a language:**
  - simplicity of the equipment demanded by the instruction set
  - clarity of built applications

Below Your Program

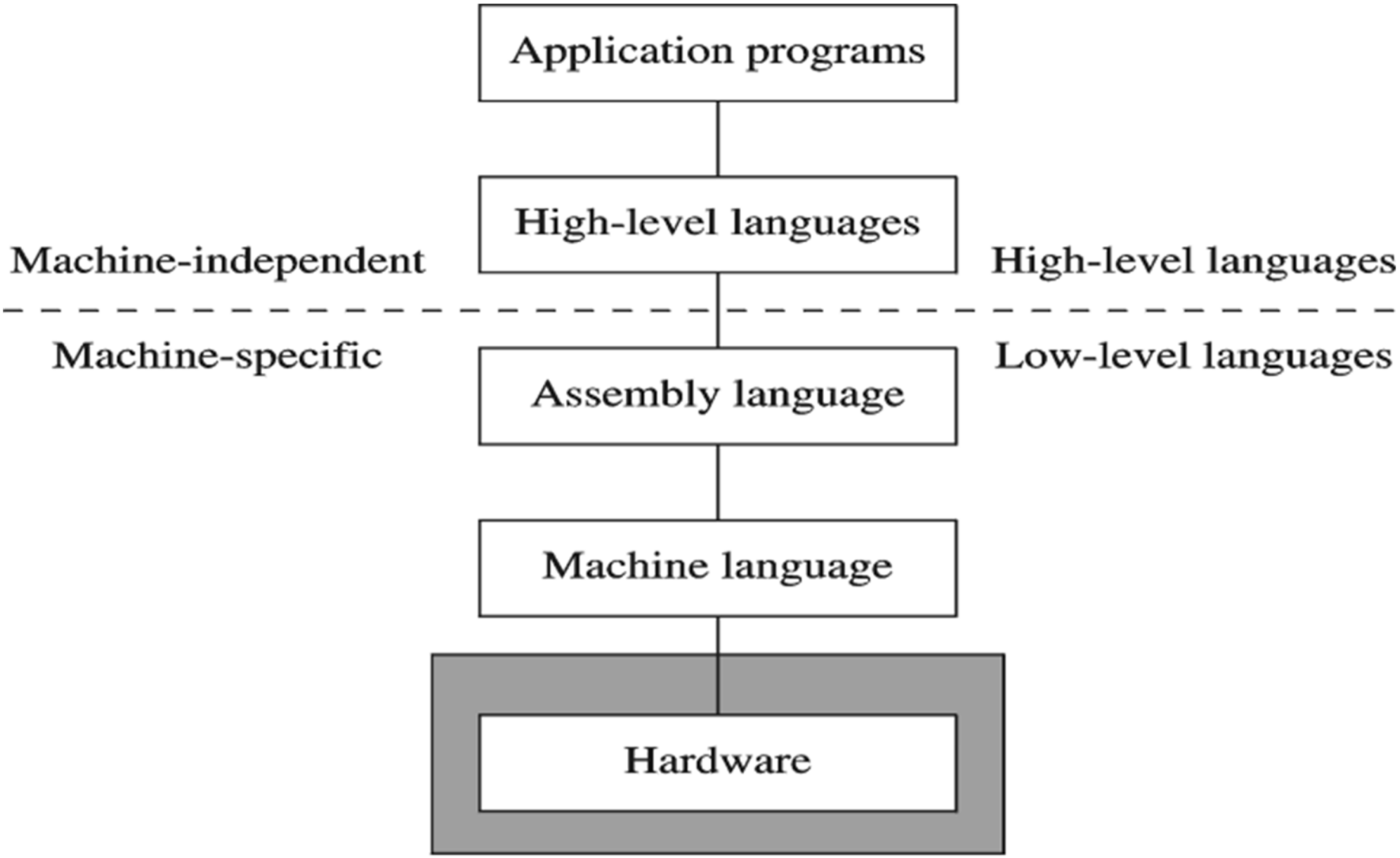
- A typical application may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application
- The hardware in a computer can only execute extremely simple low-level instructions
- To go from a complex application to the simple instructions involves several layers of software that interpret high-level operations into simple computer instructions
- These layers of software are organized in a hierarchical fashion, with applications being the outermost ring and a variety of systems software sitting between the hardware and the applications software



Below Your Program (cont.)

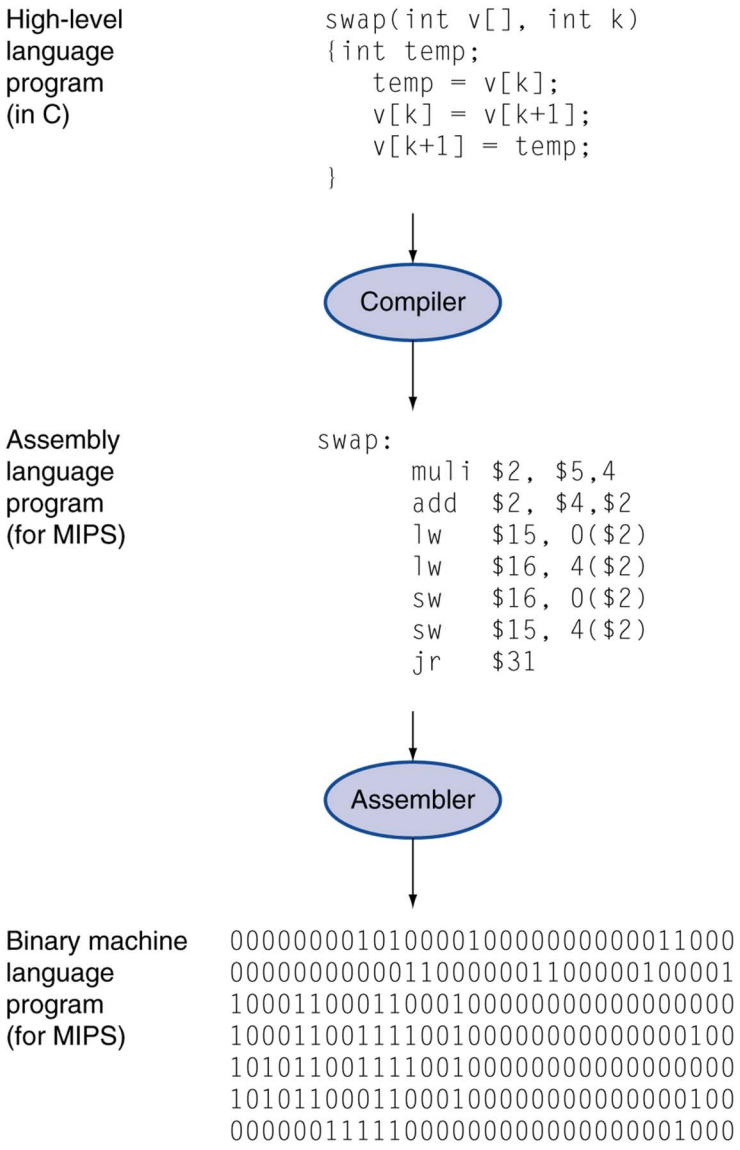
- Systems software provides services that are commonly useful
- Two types of systems software are central to every computer system today: operating systems (OSs) and compilers
- The OS is a supervising program that manages the resources of a computer for the benefit of the programs that run on that computer
- An OS interfaces between a user's program and the hardware
- An OS provides a variety of services and supervisory functions, like:
  - Handling basic input and output operations
  - Allocating storage and memory
  - Providing for protected sharing of the computer among multiple applications using it simultaneously
- Compilers translate a program written in a high-level language (HLL) (e.g., C, C++, Java) into assembly language instructions that the hardware can execute

A Hierarchy of Languages



From a High-Level Language to the Hardware Language

- Instructions and data are just collection of **bi**nary digi**ts** (**bits**)
- **Instructions** are individual commands that computers understand and obey
- Hardware executes **machine instructions** (extremely simple low-level instructions)
- Machine language is the binary representation of machine instructions
- **Assembly language** symbolically represents machine instructions
- An assembler translates a symbolic version of a machine instruction into its binary version
- A high-level programming language is composed of words and algebraic notation that can be translated by a compiler into assembly language



Abstraction

- Abstraction is the use of hierarchical layers in structuring both hardware and software
- Abstraction is a model that renders lower-level details of computer systems temporarily invisible to facilitate design of sophisticated systems
- An abstraction omits unneeded details to cope with complexity
- Lower-level details are hidden to offer a simpler model at higher levels
- Digging into the depths reveals more information
- The principle of abstraction is the way both hardware designers and software designers cope with the complexity of computer systems

Instruction Set Architecture (ISA)

- **Instruction Set Architecture (ISA)**, or simply **Architecture**, of a computer is an abstract interface between the hardware and the lowest-level software

- ISA is a key interface between the levels of abstraction
- ISA is one of the most important abstractions

- ISA encompasses all the information necessary for programmers to write a machine language program that will run correctly, including instructions, registers, memory access, I/O devices, etc.

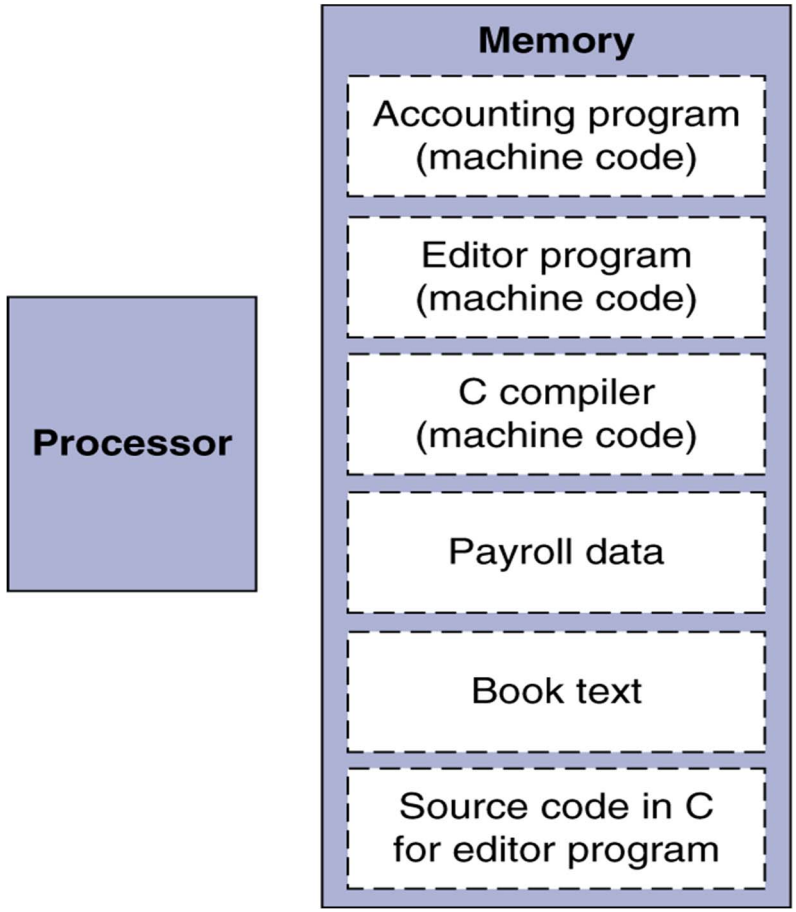
- ISA standardizes instructions, machine language bit patterns, etc.
- ISA allows computer designers to talk about functions independently from the hardware that performs them

- The ISA abstract interface enables many implementations (presumably varying in cost and performance) to run identical software (same architecture)

The Stored-Program Concept

- Instructions are represented as numbers
- Programs are stored in memory as numbers to be read or written just like data
- Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer system
- Memory technology needed for data can also be used for programs

- **Principles:**
  - Use of instructions that are indistinguishable from numbers
  - Use of alterable memory for programs



Individual Exercise (1)

- The stored-program concept, introduced in the late 1940s, brought about a significant change in how computers were designed and operated

1. What is a possible example of a nonstored-program computer?
2. and what are the problems with such a computer?
3. How can these problems be overcome by a stored-program computer?

Individual Exercise (1): Answer

1. What is a possible example of a nonstored-program computer?
   - Wired calculator
2. and what are the problems with such a computer?
   - Without a stored program, the programmer must physically configure the computer to run the desired program
   - Hence, a nonstored-program computer is one where the computer must essentially be rewired to run the program.
   - The problem with such a computer is that much time must be devoted to reprogramming it if one wants to either run another program or fix bugs in the current program
3. How can these problems be overcome by a stored-program computer?
   - The stored-program concept is important in that a programmer can quickly modify and execute a stored program, resulting in the computer being more of a general-purpose one instead of a specifically wired calculator

Operations

- MIPS assembly language notation for arithmetic operations:

```
operation r1, r2, r3        # r2 operation r3 → r1
```

- Each line of this language may contain at most one instruction
- Each MIPS arithmetic instruction performs only one operation
- Each MIPS arithmetic instruction must have **exactly** three operands
- Operand order is fixed (destination first)
- The words to the right of the sharp symbol (#) are comments
- Comments always terminate at the end of a line

- **Design principle 1: Simplicity favors regularity**
  - Requiring every arithmetic instruction to have exactly three operands and one operation keeps the hardware simple
  - Hardware for a variable number of operands is more complicated than hardware for a fixed number

Individual Exercise (2)

- Write a MIPS assembly code that places the sum of the four integer variables b, c, d, and e into integer variable a, then subtracts integer variable f from a and puts the result in integer variable g

  C code:

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

- How many MIPS assembly instructions does it take to sum the four variables?

Individual Exercise (2): Answer

- Write a MIPS assembly code that places the sum of the four integer variables b, c, d, and e into integer variable a, then subtracts integer variable f from a and puts the result in integer variable g

  C code:

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

  MIPS code:

```
add a, b, c          # b + c → a
add a, a, d          # a + d → a
add a, a, e          # a + e → a
sub g, a, f          # a - f → g
```

- How many MIPS assembly instructions does it take to sum the four variables?
  - It takes three assembly instructions to sum the four variables

Operands

- In MIPS, arithmetic instructions' operands must be registers

- MIPS has only 32 registers (limited number!)
- For MIPS, a word is 32 bits (or 4 bytes)
- MIPS registers hold 32 bits of data (a word size)

- MIPS is a general-purpose register architecture
  - MIPS registers are general-purpose registers (**GPR**s)
  - MIPS registers can be used for addresses or data with any instruction
  - x86 registers have dedicated uses

- Registers are visible to the programmer
- It is the compiler's job to associate program variables with registers
- Effective use of registers is a key to program performance

- **Design principle 2: Smaller is faster**
  - A very large number of registers may increase the clock cycle time since it takes electronic signals longer when they must travel farther

Individual Exercise (3)

- The following C statements contain the six integer variables e, f, g, h, i, j:

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

- Suppose that the compiler associates variables e, f, g, h, i, and j with registers $s0 through $s5, respectively

- What is the compiled MIPS assembly code?

Individual Exercise (3): Answer

- The following C statements contain the six integer variables e, f, g, h, i, j:

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

- Suppose that the compiler associates variables e, f, g, h, i, and j with registers $s0 through $s5, respectively

- What is the compiled MIPS assembly code?

MIPS code:

```
add $t0, $s2, $s3    # $t0 is a temporary register
add $t1, $s4, $s5    # $t1 is a temporary register
sub $s1, $t0, $t1
add $s0, $s1, $zero  # copy f ($s1) to e ($s0)
```

Registers

- MIPS convention for naming registers: use two character names (except for $zero) following a dollar sign
- MIPS register **$zero** always maps to zero

Name	Register number (decimal)	Usage	Preserve on call?
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	procedure return values and expression evaluation	no
$a0-$a3	4-7	procedure arguments (parameters)	no
$t0-$t7	8-15	temporary registers	no
$s0-$s7	16-23	general purpose saved registers	yes
$t8-$t9	24-25	more temporary registers	no
$k0-$k1	26-27	reserved for the OS	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	procedure return address	yes

Memory Organization

- Memory is viewed as a large, single-dimension array, with an address
- A memory address acts as an index to the array, starting at 0
- MIPS memory could be viewed as:
  - 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
  - 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$

	bytes
0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
...	

	words
0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

- **MIPS uses a byte addressing:** a memory index points to a byte of memory
- **Word alignment restriction:** words must start at addresses that are multiple of 4 in MIPS
  - the least 2 significant bits of a word address must be "00"
  - Note that we need to convert an array (of words) index into a byte offset by multiplying it by 4 (shift left by 2 bits)!

Memory Organization (cont.)

- Complex data structures (like arrays and structures) require more space than there are registers in a machine
  - Such data structures are kept in memory and moved to registers
- When programs have more variables than machines have registers, the compiler tries to keep the most frequently used variables in registers and places the rest in memory

- **Spilling registers:** the process of putting less commonly used variables (or those needed later) into memory

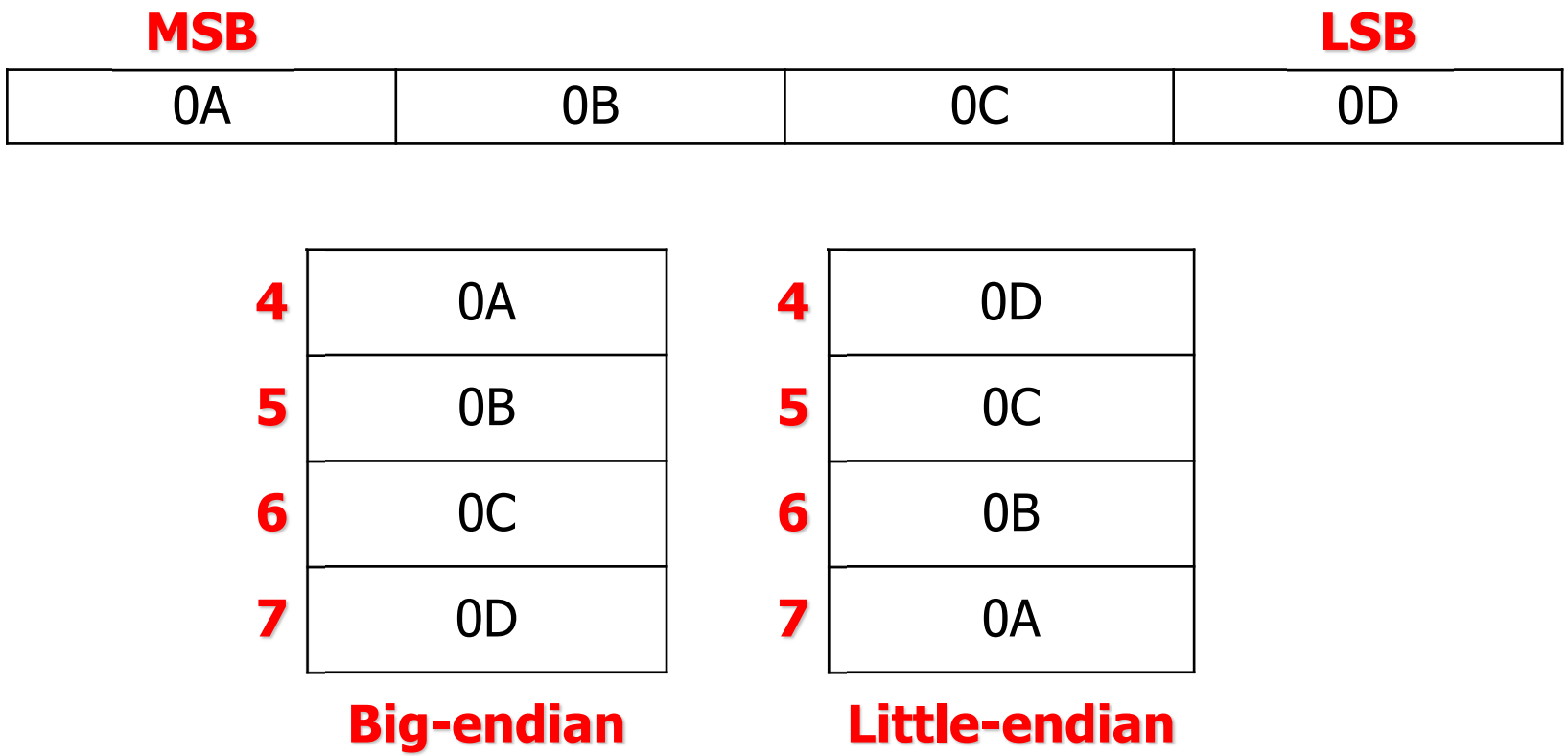
- **Endianness (byte order):** the ordering used to store multi-byte integers as bytes in memory relative to a given memory addressing scheme
- MIPS is a **big-endian** machine
  - it uses the address of the leftmost byte as the word address
- Pentium is a **little-endian** machine
  - it uses the address of the rightmost byte as the word address

Individual Exercise (4)

- Show how the 32-bit hex number 0A0B0C0D is saved in the memory of big-endian and little-endian machines, starting at location 4 in memory

Individual Exercise (4): Answer

- Show how the 32-bit hex number 0A0B0C0D is saved in the memory of big-endian and little-endian machines, starting at location 4 in memory



Data Transfer Instructions

- **Data transfer instructions:** transfer data between memory and registers using **offset** and **base register** addressing
  - **load word** (`lw`): copies a word from memory to a register
  - **store word** (`sw`): copies a word from a register to memory

```
lw $s0, c ($s1)  # Memory [$s1 + c] → $s0
sw $s0, c ($s1)  # $s0 → Memory [$s1 + c]
                 # $s1 is the base register
                 # constant c is the offset
                 # $s1 + c must be divisible by 4
```

- **The base register** holds the starting address of the array and the **constant** selects the desired array element (**offset**)
- Traditionally, **the offset** was put in the register (called index register) and the base was supplied as the constant
- Today, the base address of the array is passed in a register since it will not fit in the constant (memories became much larger)

Data Transfer Instructions (cont.)

- Arithmetic operations read and operate on two registers and write one
- Data transfer operations read or write only one register without operating on it
- MIPS memory is only accessed through **loads** and **stores**:
  - This is why MIPS, like all other RISC machines, are called **load/store architectures**
  - MIPS arithmetic operands are registers only, not memory!
  - Registers take less time and have higher throughput than memory
  - Data in registers are both faster to access and simpler to use than memory
  - To achieve highest performance, compilers must use registers efficiently

Individual Exercise (5)

- Lets assume that the compiler has associated integer variables h and i with registers $s2 and $s4, respectively
- Lets also assume that the base address of the integer array A is in register $s3

- **Compile** the following C statement into MIPS assembly code:

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

Individual Exercise (5): Answer

- Lets assume that the compiler has associated integer variables h and i with registers $s2 and $s4, respectively
- Lets also assume that the base address of the integer array A is in register $s3

- Compile the following C statement into MIPS assembly code:

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

```
lw   $t0, 32 ($s3)    # Memory [$s3 + 8*4] → $t0
add  $t0, $s2, $t0    # h + A[8] → $t0
add  $t1, $s4, $s4    # $t1 = 2*i
add  $t1, $t1, $t1    # $t1 = 4*i
add  $t1, $t1, $s3    # $t1 = $s3 + 4*i (address of A[i])
sw   $t0, 0 ($t1)     # $t0 → Memory [$t1]
```

  - This is an example for compilation using static and variable array indices

Data Transfer Instructions (cont.)

- **MIPS provides instructions to load and store bytes and halfwords**

```
lb  $t0, 0 ($s0)     # load signed byte
sb  $t0, 0 ($s0)     # store byte
lh  $t0, 0 ($s0)     # load signed halfword
sh  $t0, 0 ($s0)     # store halfword
```

  - `lb`: copies a **signed** byte from memory to a register rightmost byte. The byte is **sign-extended** to the 32 bits of the register
  - `sb`: copies a byte from a register rightmost byte to memory
  - `lh`: copies a **signed** halfword from memory to a register's rightmost halfword. The halfword is **sign-extended** to the 32 bits of the register
  - `sh`: copies a halfword from a register's rightmost halfword to memory

Small Constant or Immediate Operands

- **There are two ways to handle small constants:**
  - A slower way is to place constants in memory and then load them

```
lw   $t0, const_4 ($s1)    # load constant 4 to $t0
# $s1 + const_4 is the memory address of constant 4
add  $s3, $s3, $t0         # $s3 = $s3 + 4
```

  - A faster way is to use the arithmetic & logical instruction version in which one operand is a constant (immediate) of up to 16 bits

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

- **Design principle 3: Make the common case fast**
  - Small constants are used quite frequently (> 50% of operands)

Group Exercise (6)

- Why doesn't MIPS have a subtract immediate instruction?

Group Exercise (6): Answer

- Why doesn't MIPS have a subtract immediate instruction?
- Since MIPS includes add immediate and since immediates can be positive or negative, subtract immediate would be redundant
- In other words, since MIPS supports negative constants, there is no need for subtract immediate in MIPS
- For example:
  - `addi $t0, $t1, -8` # $(-2^{15} \rightarrow 2^{15} - 1)$

Logical (Bitwise) Instructions

Shift left logical	`sll n`	Move all the bits in a word to the left by n bits, filling the emptied bits with 0s (equivalent to **signed or unsigned multiplication** by 2^n)
Shift right logical	`srl n`	Move all the bits in a word to the right by n bits, filling the emptied bits with 0s (equivalent to **unsigned division** by 2^n, **rounding towards 0**)
Shift right arithmetic	`sra n`	Move all the bits in a word to the right by n bits, filling the emptied bits with the sign bit (equivalent to **signed division** by 2^n, **rounding down towards -∞**)
Bit-by-bit AND	`and`	bit-by-bit register AND register
Bit-by-bit AND immediate	`andi`	bit-by-bit register AND small constant
Bit-by-bit OR	`or`	bit-by-bit register OR register
Bit-by-bit OR immediate	`ori`	bit-by-bit register OR small constant
Bit-by-bit NOR	`nor`	bit-by-bit register NOR register
Bit-by-bit XOR	`xor`	bit-by-bit register XOR register
Bit-by-bit XOR immediate	`xori`	bit-by-bit register XOR small constant

Logical (Bitwise) Instructions (cont.)

- `not` is implemented using a `nor` with one operand being $zero for regularity
- To reset/conceal (set to 0) some bits, `and` can be used with a **mask**
- To set some bits to 1, `or` can be used with a **mask**

Group Exercise (7)

- Consider the following register contents:

```
$s0 = 1001 0000 1100 0101 1111 1100 0011 1010
$s1 = 0101 0110 1001 1111 0000 0011 1100 0101
```

- What would the value in register $t0 be after executing each of the following instructions?
  - `sll $t0, $s0, 4`
  - `srl $t0, $s0, 4`
  - `sra $t0, $s0, 4`
  - `and $t0, $s0, $s1`
  - `or  $t0, $s0, $s1`
  - `nor $t0, $s0, $s1`
  - `xor $t0, $s0, $s1`

Group Exercise (7): Answer

- Consider the following register contents:

```
$s0 = 1001 0000 1100 0101 1111 1100 0011 1010
$s1 = 0101 0110 1001 1111 0000 0011 1100 0101
```

- What would the value in register $t0 be after executing each of the following instructions?

`sll $t0, $s0, 4`	0000 1100 0101 1111 1100 0011 1010 **0000**
`srl $t0, $s0, 4`	**0000** 1001 0000 1100 0101 1111 1100 0011
`sra $t0, $s0, 4`	**1111 1**001 0000 1100 0101 1111 1100 0011
`and $t0, $s0, $s1`	0001 0000 1000 0101 0000 0000 0000 0000
`or $t0, $s0, $s1`	1101 0110 1101 1111 1111 1111 1111 1111
`nor $t0, $s0, $s1`	0010 1001 0010 0000 0000 0000 0000 0000
`xor $t0, $s0, $s1`	1100 0110 0101 1010 1111 1111 1111 1111

Group Exercise (8)

- Write MIPS assembly instructions to
  - Set bits 1, 5, and 12 of $s0 to 0
  - Set bits 2, 3, 11 of $s0 to 1
  - Invert all bits of $s0
  - Invert the least significant bit of $s0
  - Hint: you can use register $t1 for a mask

Group Exercise (8): Answer

- Write MIPS assembly instructions to
  - Set bits 1, 5, and 12 of $s0 to 0

```
$t1 = 1111 1111 1111 1111 1110 1111 1101 1101
and $s0, $s0, $t1
```

  - Set bits 2, 3, 11 of $s0 to 1

```
$t1 = 0000 0000 0000 0000 0000 1000 0000 1100
or $s0, $s0, $t1
```

  - Invert all bits of $s0

```
nor $s0, $s0, $zero
```

  - Invert the least significant bit of $s0

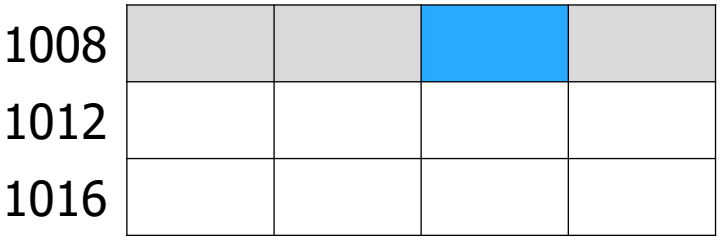
```
$t1 = 0000 0000 0000 0000 0000 0000 0000 0001
xor $s0, $s0, $t1
```

Individual Exercise (9)

- Write MIPS assembly instructions to load one unsigned byte from memory address 1010 into register $s0
- Do not use MIPS `lb` or `lh` instructions
- Assume that register $t1 has the value 1016
- Solve using two approaches to get the required byte:
  1. Using a shift and `and` (with a mask) operations
  2. Using only shift operations

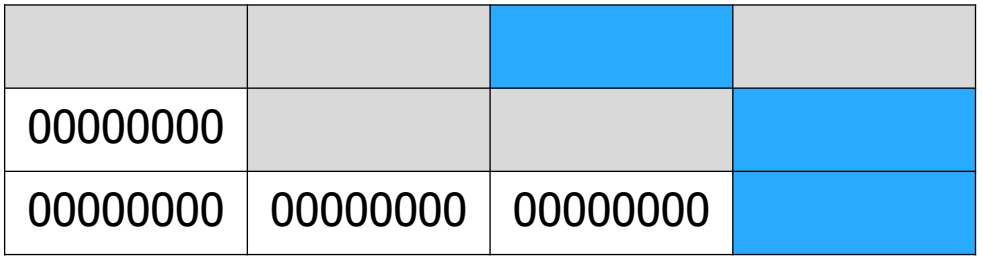
Individual Exercise (9): Answer

- lw can only read complete words (32 bits) from memory
- Read a complete word then:



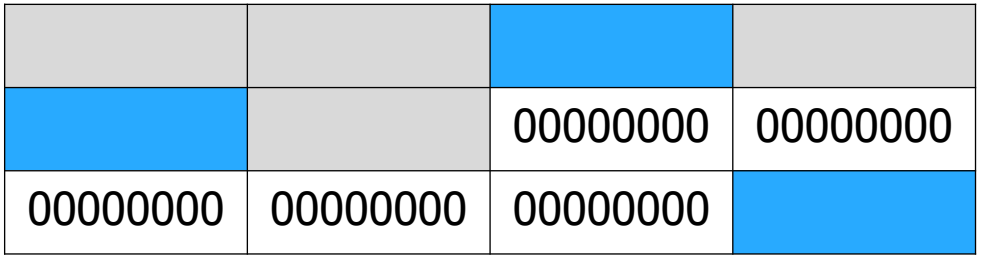
1. Use shift and `and` (with a mask) operations

```
lw   $s0, -8 ($t1)
srl  $s0, $s0, 8
andi $s0, $s0, 255
```



2. Use shift operations to get the required byte

```
lw   $s0, -8 ($t1)
sll  $s0, $s0, 16
srl  $s0, $s0, 24
```



Sign and Zero Extension of Immediates

- Creating 32-bit constants from 16-bit immediates needs care!

- Immediates in arithmetic instructions are **signed** ($-2^{15} \rightarrow 2^{15} - 1$)
  - Before performing an arithmetic instruction with an immediate operand, sign-extend the immediate operand
  - **Sign extension:** extends the 16-bit immediate field of the instruction to a 32-bit word by copying the leftmost (sign) bit of the constant into the upper 16 bits of the word

- Immediates in logical instructions are **unsigned** ($0 \rightarrow 2^{16} - 1$)
  - Before performing a logical instruction with an immediate operand, zero-extend the immediate operand
  - **Zero extension:** extends the 16-bit immediate field of the instruction to a 32-bit word by loading 0s into the upper 16 bits of the word

32-Bit Immediate Operands

- Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger

- To handle constants larger than 16 bits:
  - Use the instruction **load upper immediate** (`lui`) to set the upper 16 bits of a constant in a register
    - filling the lower 16 bits with 0s
  - Then, use a subsequent **OR immediate** (`ori`) instruction to specify the lower 16 bits of the constant

Individual Exercise (10)

- What is the MIPS assembly code to load the following 32-bit constant into register $s0?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Individual Exercise (10): Answer

- What is the MIPS assembly code to load the following 32-bit constant into register $s0?

61	2304
0000 0000 0011 1101	0000 1001 0000 0000

```
lui   $s0, 61                  # 61 is 0000 0000 0011 1101
```

  - The value of register $s0 afterward is

0000 0000 0011 1101	0000 0000 0000 0000

```
ori   $s0, $s0, 2304           # 2304 is 0000 1001 0000 0000
```

  - The value of register $s0 now is

0000 0000 0011 1101	0000 1001 0000 0000

Individual Exercise (11)

- Lets assume that the compiler has associated integer variables x, y, and z with registers $s0, $s1, and $s2, respectively

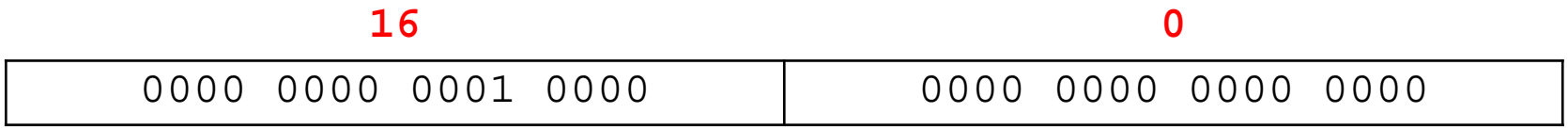
- **Compile** the following C code segment into MIPS assembly code:

```
z = x + y;
x += 10;
y += 1048576;      /* 1048576 = 2^20 */
z += 1048581;      /* 1048581 = 2^20 + 5 */
```

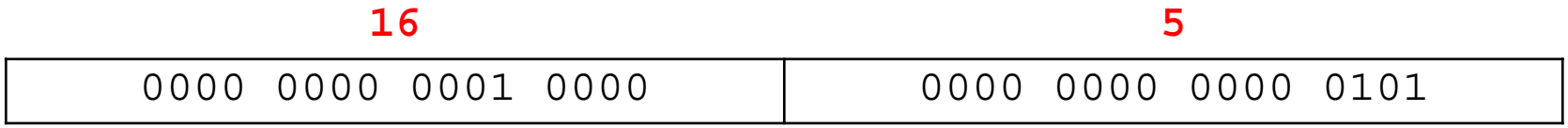
Individual Exercise (11): Answer

- Lets assume that the compiler has associated integer variables x, y, and z with registers $s0, $s1, and $s2, respectively
- Compile the following C code segment into MIPS assembly code:

$1048576 = 2^{20}$



$1048581 = 2^{20} + 5$




```
z = x + y;
x += 10;
y += 1048576;

z += 1048581;
```

```
add   $s2, $s0, $s1
addi  $s0, $s0, 10
lui   $t0, 16
add   $s1, $s1, $t0
ori   $t0, $t0, 5
add   $s2, $s2, $t0
```

Individual Exercise (12)

- State whether the following statement is correct or not. If it is incorrect, give a **counterexample**

  "We can use an `addi` instruction (instead of `ori`) after an `lui` one to save 32-bit immediate constants into registers"

Individual Exercise (12): Answer

- State whether the following statement is correct or not. If it is incorrect, give a counterexample

  "We can use an `addi` instruction (instead of `ori`) after an `lui` one to save 32-bit immediate constants into registers"

- Incorrect!

- The instruction `addi` sign-extends the 16-bit immediate field of the instruction to a 32-bit word

- The instruction `ori` zero-extends the 16-bit immediate field of the instruction to a 32-bit word and hence is used by the assembler in conjunction with `lui` to create 32-bit constants

Individual Exercise (12): Answer (cont.)

- **Counterexample**: consider the constant: 0x80018001

0x8001	0x8001
1000 0000 0000 0001	1000 0000 0000 0001

```
lui  $s0, 0x8001          # 0x8001 is 1000 0000 0000 0001
```

  - The value of register $s0 afterward is

1000 0000 0000 0001	0000 0000 0000 0000

```
addi $s0, $s0, 0x8001     # 0x8001 is 1000 0000 0000 0001
```

  - The value of register $s0 now is (incorrect)

1000 0000 0000 0001	0000 0000 0000 0000
+ 1111 1111 1111 1111 ← - - - -	1000 0000 0000 0001
= 1000 0000 0000 0000	1000 0000 0000 0001

Individual Exercise (12): Answer (cont.)

- However,

```
ori  $s0, $s0, 0x8001   # 0x8001 is 1000 0000 0000 0001
```

  - The value of register $s0 now is

1000 0000 0000 0001	0000 0000 0000 0000
OR 0000 0000 0000 0000	1000 0000 0000 0001
= 1000 0000 0000 0001	1000 0000 0000 0001

MIPS Simulators

- You can practice MIPS code assembly using MIPS simulators
- You can use them to verify your assignment solutions

- **QtSpim MIPS Simulator**
  - Download the source from links at:
    - http://spimsimulator.sourceforge.net/

      or
    - http://sourceforge.net/projects/spimsimulator/files/

- **MARS MIPS Simulator**
  - Download the source from the link at:
    - http://courses.missouristate.edu/KenVollmar/mars/

Individual Exercise (13)

- Construct the QtSpim code used to simulate the compiled MIPS assembly code in the Exercise (3)?

Individual Exercise (13): Answer

- **QtSpim Code**

```
.data # variable declarations
e:      .word   0
f:      .word   0
g:      .word   5
h:      .word   7
i:      .word   2
jj:     .word   3
```

```
.text    # instructions

main:    # start of code
lw       $s2, g
lw       $s3, h
lw       $s4, i
lw       $s5, j

add      $t0, $s2, $s3
add      $t1, $s4, $s5
sub      $s1, $t0, $t1
add      $s0, $s1, $zero

sw       $s1, f
sw       $s0, e

li       $v0, 10
syscall  # ends execution
.end
```

Instruction Formats

- Instructions are kept in the computer as a series of bits and maybe represented as numbers

- Each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction

- **Machine language:** is the numeric version of instructions
- **Machine code:** a sequence of machine instructions
- **Instruction format:** the layout of the instruction fields and bits
- **Instruction field:** a machine language instruction segment

- All MIPS instructions take exactly 32 bits (the same as a data word)
- This agrees with **design principle 1: Simplicity favors regularity!**

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
R-format	op	rs	rt	rd	shamt	funct
I-format	op	rs	rt	16-bit immediate/address		
J-format	op	26-bit address				

- **Register format: R-format**
  - Used by arithmetic and logical instructions
- **Immediate format: I-format**
  - Used by data transfer instructions
  - Used by instructions that have immediate operands
  - Used by relative-address branching
- **Jump format: J-format**
  - Used by absolute-jump instructions

Instruction Formats (cont.)

- **op:** basic operation of the instruction, traditionally called **opcode**
- **rs:** the first register source operand
- **rt:** the second register source operand in R-format. This field sometimes specifies a destination register in I-format
- **rd:** the register destination operand that gets the operation result
- **shamt:** shift amount, used in shift instructions
- **funct:** function code that selects the specific variant of the operation in the op field in R-format
- **16-bit immediate (constant)/address:** a **signed** numerical field
- **26-bit address**

- **Different formats are distinguished by the values in the first field**
  - Each format is assigned a distinct set of values in the first field (**op**)
  - The hardware knows whether to treat the last half of the instruction as three fields (R-format) or as a single field (I-format)

- **The 16 bit immediate (constant)/address signed field means:**
  - `lw` and `sw` can load/store any word within a region of $\pm 2^{15}$ bytes or $\pm 2^{13}$ words of the address in the base register
  - `addi` is limited to constants no larger than $\pm 2^{15}$

Instruction Formats (cont.)

- **Design principle 4: Good design demands good compromises**
  - Compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length

- We have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format

- MIPS keeps all instructions the same length, providing different kinds of instruction formats for different kinds of instructions

- **Hardware complexity can be reduced by keeping formats similar**
  - The first three fields of the R-format and I-format are the same size and have the same names
  - MIPS keep register fields in the same place in each instruction format
  - The length of the fourth field in I-format is equal to the sum of the lengths of the last three fields of R-format
  - There is a similarity between the binary representations of related instructions (e.g., `lw` and `sw`), which simplifies the hardware design

Instruction Formats (cont.)

Instruction	Format	op	funct	Instruction	Format	op	funct
add	R	0_{10}	32_{10}	or	R	0	37
addi	I	8		ori	I	13	
and	R	0	36	sb	I	40	
andi	I	12		sh	I	41	
beq	I	4		sll	R	0	0
bne	I	5		slt	R	0	42
j	J	2		slti	I	10	
jal	J	3		sra	R	0	3
jr	R	0	8	srl	R	0	2
lb	I	32		sub	R	0	34
lh	I	33		sw	I	43	
lui	I	15		xor	R	0	38
lw	I	35		xori	I	14	
nor	R	0	39				

- Refer to the **MIPS reference data card** in the internal front cover of the book for the **op** and **funct** values for all instructions

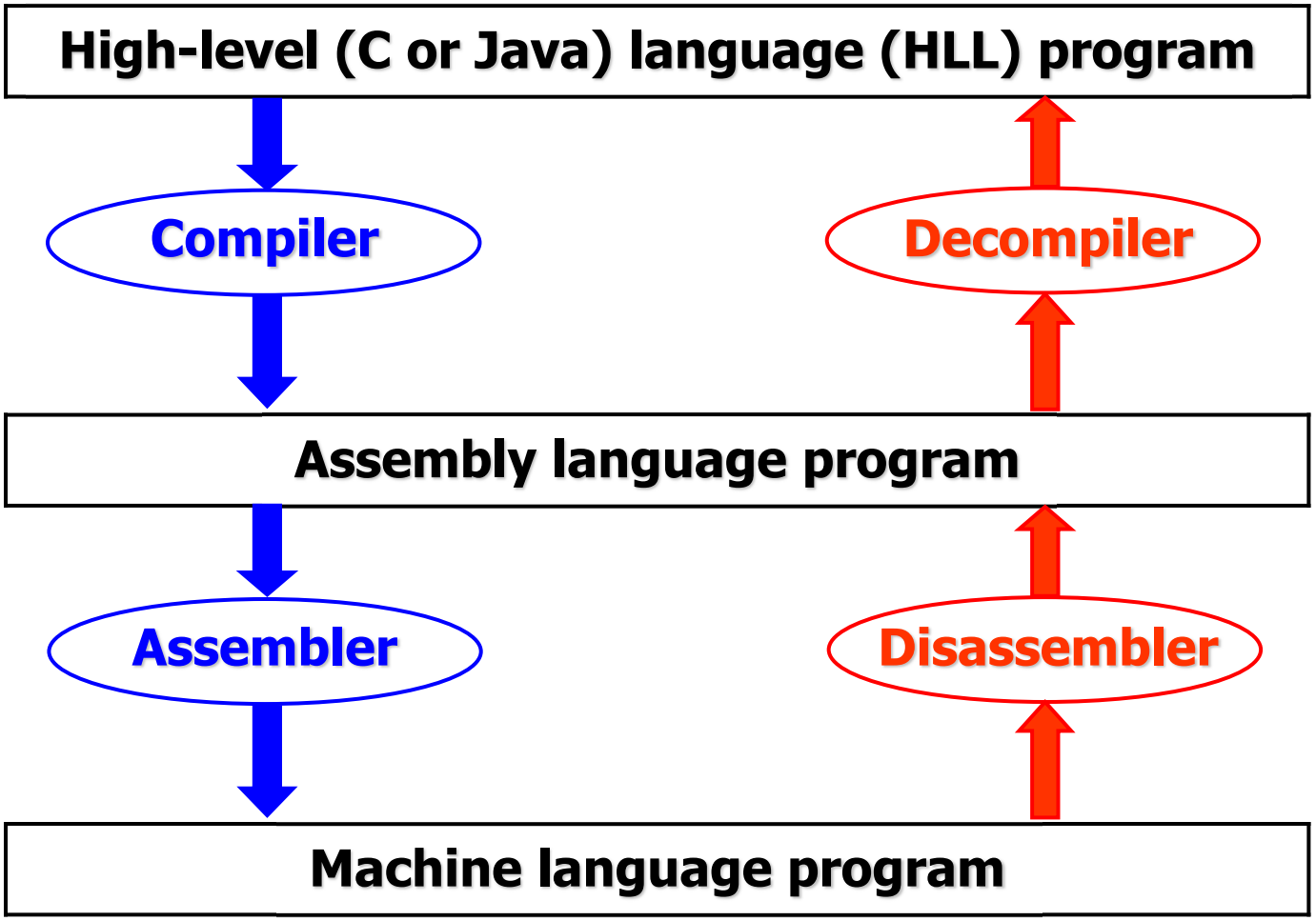
Instruction Formats (cont.)

R-format instructions	
Arithmetic and logic	
add	$t1, $t2, $t3
sub	$t1, $t2, $t3
and	$t1, $t2, $t3
or	$t1, $t2, $t3
xor	$t1, $t2, $t3
nor	$t1, $t2, $t3
slt	$t1, $t2, $t3
Shift	
sll	$t1, $t2, 5
sra/srl	$t1, $t2, 5
Jump	
jr	$ra

J-format instructions	
Jump	
j	L1
jal	L1

I-format instructions	
Arithmetic and logic	
addi	$t1, $t2, 15
andi	$t1, $t2, 15
ori	$t1, $t2, 15
xori	$t1, $t2, 15
slti	$t1, $t2, 15
lui	$t1, 15
Load and store	
lw	$t1, 15 ($t2)
lh	$t1, 15 ($t2)
lb	$t1, 15 ($t2)
sw	$t1, 15 ($t2)
sh	$t1, 15 ($t2)
sb	$t1, 15 ($t2)
Decision making	
beq	$t1, $t2, L1
bne	$t1, $t2, L1

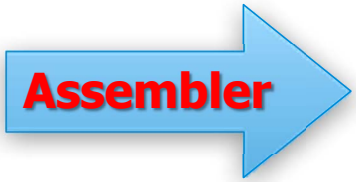
Program Translation Hierarchy



Assemblers

- The assembler is a computer program, which translates assembly language to machine code, saved in an object file
- Machine code is the interface between software and hardware
- There is a one-to-one correspondence between assembly instructions and machine code

```
Loop:  lw    $t3, 0($t0)
       lw    $t4, 4($t0)
       add   $t2, $t3, $t4
       sw    $t2, 8($t0)
       addi  $t0, $t0, 4
       addi  $t1, $t1, -1
       bne   $t1, $zero, loop
```




```
0x8D0B 0000
0x8D0C 0004
0x016C 5020
0xAD0A 0008
0x2108 0004
0x2129 FFFF
0x1520 FFF9
```

Assembly program
(text file)
source code

Machine code
(binary)
object code

How to Assemble Assembly Instructions

1. Decide the format (R, I, or J) of the instruction
2. Determine the value of each instruction field (component)
3. Convert each field value to binary
4. Put together the full binary code of the instruction
5. Convert the instruction binary code to hexadecimal

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
R-format	op	rs	rt	rd	shamt	funct

Fields of R-format instructions		
op	6 bits	Always zero
rs	5 bits	1st argument register
rt	5 bits	2nd argument register
rd	5 bits	Destination register
shamt	5 bits	Used only in shift instructions
funct	6 bits	Code for the operation to perform
Note: the destination register is the third in the machine code		

Individual Exercise (14)

- Show the real MIPS machine language version for the instruction represented symbolically as

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

  - Registers have numbers: `$t0 = 8, $s1 = 17, $s2 = 18`

Individual Exercise (14): Answer

- Show the real MIPS machine language version for the instruction represented symbolically as

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

  - Registers have numbers: `$t0 = 8, $s1 = 17, $s2 = 18`

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

Machine code: `00000010001100100100000000100000 = 0x02324020`

  - The **first** and **last** fields (containing 0 and 32) in combination tell the MIPS machine that this instruction performs addition
  - The **fifth** field is not used in this instruction (set to 0)

Individual Exercise (15)

- **Assemble** the following MIPS instruction into real MIPS machine code:
  - `sll $t2, $s0, 4`
  - Registers have numbers: `$t2 = 10, $s0 = 16`

Individual Exercise (15): Answer

- Assemble the following MIPS instruction into real MIPS machine code:
  - `sll $t2, $s0, 4`
  - Registers have numbers: `$t2 = 10, $s0 = 16`

op	rs	rt	Rd	Shamt	Funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

Machine code: `00000000000010000010100010000000 = 0x00105100`

- The encoding of `sll` is 0 in both the **op** and **funct** fields
- The **rs** field is unused in shift instructions (**sll** as well as **sra** and **srl**) and thus is set to 0

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
I-format	op	rs	rt	16-bit immediate/address		

Fields of I-format instructions		
op	6 bits	Code for the operation to perform
rs	5 bits	1st argument register
rt	5 bits	Destination or 2nd argument register
imm	16 bits	Constant (memory **offset** or **immediate**) value embedded in the instruction
Note: the destination register is the second in the machine code		

Individual Exercise (16)

- **Assemble** the following MIPS instruction into real MIPS machine code:

```
lw $t0, 32 ($s3)
```

  - Registers have numbers: `$t0 = 8, $s3 = 19`

Individual Exercise (16): Answer

- Assemble the following MIPS instruction into real MIPS machine code:

```
lw $t0, 32 ($s3)
```

  - Registers have numbers: `$t0 = 8, $s3 = 19`

Op	rs	rt	immediate/address
35	19	8	32
100011	10011	01000	0000 0000 0010 0000

Machine code: `10001110011010000000000000100000 = 0x8E680020`

- The first field (containing 35) tells the MIPS machine that this instruction performs a load word

Individual Exercise (17)

- **Assemble** the following MIPS instruction into real MIPS machine code:

```
addi $s1, $s2, 4
```

  - Registers have numbers: `$s1 = 17, $s2 = 18`

Individual Exercise (17): Answer

- Assemble the following MIPS instruction into real MIPS machine code:

```
addi $s1, $s2, 4
```

  - Registers have numbers: `$s1 = 17, $s2 = 18`

op	rs	rt	immediate/address
8	18	17	4
001000	10010	10001	0000 0000 0000 0100

MIPS machine instruction: `00100001001010001000000000000100`

MIPS machine instruction (in hex): `0x2251 0004`

Individual Exercise (18)

- **Assemble** the following MIPS instruction into real MIPS machine code:

```
lui   $t0, 255
```

  - register $t0 number is 8

Individual Exercise (18): Answer

- Assemble the following MIPS instruction into real MIPS machine code:

```
lui    $t0, 255
```

  - register $t0 number is 8

op	rs	rt	immediate/address
15	0	8	255
001111	00000	01000	0000 0000 1111 1111

Machine code: `00111100000010000000000011111111 = 0x3C0800FF`

  - The **rs** field is unused, and thus is set to 0

Group Exercise (19)

- If $t1 has the base of the integer array A and that $s2 corresponds to integer variable h, **compile** the following C assignment statement into MIPS assembly code. Then, **assemble** the MIPS instructions into real MIPS machine code. Use temporary register $t0, if needed

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

  - Registers have numbers: `$t0 = 8, $t1 = 9, $s2 = 18`

Group Exercise (19): Answer

instruction	op	rs	rt	rd	address/ shamt	funct
lw $t0, 1200 ($t1)	35	9	8	1200		
add $t0, $s2, $t0	0	18	8	8	0	32
sw $t0, 1200 ($t1)	43	9	8	1200		

lw $t0, 1200 ($t1)	100011	01001	01000	0000 0100 1011 0000		
add $t0, $s2, $t0	000000	10010	01000	01000	00000	100000
sw $t0, 1200 ($t1)	101011	01001	01000	0000 0100 1011 0000		

Machine code: `10001101001010000000010010110000 = 0x8D2804B0`

Machine code: `00000010010010000100000000100000 = 0x02484020`

Machine code: `10101101001010000000010010110000 = 0xAD2804B0`

MIPS machine instruction (in hex): `0x8D28 04B0 0248 4020 AD28 04B0`

Individual Exercise (20)

- **Disassemble** the following MIPS machine instruction to its equivalent MIPS assembly code

```
0x00 AF 80 20

In binary:

0000 0000 1010 1111 1000 0000 0010 0000
```

Individual Exercise (20): Answer

- Disassemble the following MIPS machine instruction to its equivalent MIPS assembly code

```
0x00 AF 80 20

In binary: 0000 0000 1010 1111 1000 0000 0010 0000
```

  - The opcode field (bits 31-26) is 0, so it is an R-format instruction:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000
0	5	15	16	0	32

  - So, the assembly instruction is:

```
add  $s0, $a1, $t7
```

Home Exercise (21)

- "**Regularity**" motivates many features of the MIPS instruction set

- Give three examples supporting this sentence!

Decision Making Instructions

- Decision making instructions alter the control flow
- Decisions can be done by:
  - Choosing from two alternatives: **if-else** (may be combined with **goto**)
  - Iterating a computation: **loops** (e.g., **while**, **do**, **for**, etc.)
  - Selecting one of many alternatives: **case/switch**

- **MIPS conditional branch instructions:**

```
beq $t0, $t1, Label  # branch if equal (I-format)
bne $t0, $t1, Label  # branch if not equal (I-format)
```

- **MIPS unconditional branch instructions:**

```
j   Label # Label is a word address (J-format)
jr  $t0   # jump register (to address in $t0) (R-format)
```

- **A basic block**
  - a sequence of instructions without branches, except possibly at the end, and without branch targets, or branch labels, except possibly at the beginning, which is fundamental to compilation!

Individual Exercise (22)

- Assuming that the five integer variables f through j correspond to the five registers $s0 through $s4

- What is the compiled MIPS assembly code for the following C code segment?

```
      if (i == j)
         goto L1;
      f = g + h;
L1:   f = f - i;
```

Individual Exercise (22): Answer

- Assuming that the five integer variables f through j correspond to the five registers $s0 through $s4

- What is the compiled MIPS assembly code for the following C code segment?

```
     if (i == j)
        goto L1;
     f = g + h;
L1:  f = f - i;
```

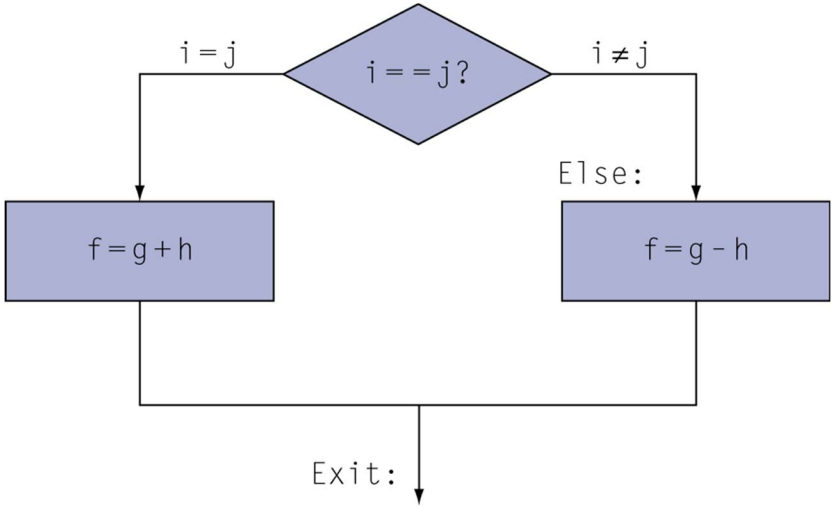
```
     beq $s3, $s4, L1
     add $s0, $s1, $s2
L1:  sub $s0, $s0, $s3
```

Individual Exercise (23)

- Assuming that the five integer variables f through j correspond to the five registers $s0 through $s4

- What is the compiled MIPS assembly code for the following C code segment?

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

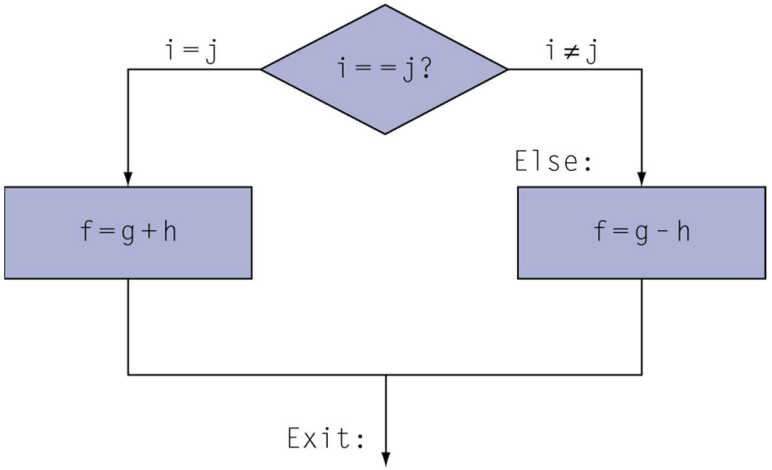


Individual Exercise (23): Answer

- Assuming that the five integer variables f through j correspond to the five registers $s0 through $s4

- What is the compiled MIPS assembly code for the following C code segment?

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




```
      bne  $s3, $s4, Else  # i ≠ j
      add  $s0, $s1, $s2   # f = g + h
      j    Exit            # Do not execute Else
Else: sub  $s0, $s1, $s2   # f = g - h
Exit:
```

Decision Making Instructions (cont.)

- **Register compare:**

```
slt   $t0, $s3, $s4   # Set on less than (R-format)
                      # Sets $t0 to 1 if $s3 < $s4
                      # else sets it to 0
slti $t0, $s2, 10     # slt immediate (I-format)
                      # Sets $t0 to 1 if $s2 < 10
                      # else sets it to 0
```

- **Compare and branch:**
  - MIPS architecture does not include a single compare and branch instruction because it is too complicated
  - Either it would stretch the clock cycle time or it would take extra clock cycles per instruction
  - Two faster instructions are more useful

Group Exercise (24)

- **Pseudoinstructions:**
  - Common variations of assembly instructions that are not part of the instruction set
  - Often appear in MIPS programs and are treated as regular ones
  - Their appearance in assembly language simplifies programming
  - The assembler produces a minimal sequence of actual MIPS instructions to accomplish their operations
  - The assembler uses the $at register to accomplish this, if needed
  - The hardware need not implement these instructions

- Use MIPS instructions to implement the following pseudoinstructions:

```
clear   $s3                 # 0 → $s3
move    $s3, $s4            # $s4 → $s3
not     $s3, $s4            # not($s4) → $s3
blt     $s3, $s4, Less      # branch on less than
```

Group Exercise (24): Answer

- Use MIPS instructions to implement the following pseudoinstructions:

```
-  clear  $s3                    # 0 → $s3

   add    $s3, $zero, $zero      # Example 1 on using $zero

-  move   $s3, $s4               # $s4 → $s3

   add    $s3, $s4, $zero        # Example 2 on using $zero

-  not    $s3, $s4               # not($s4) → $s3

   nor    $s3, $s4, $zero        # Example 3 on using $zero

-  blt    $s3, $s4, Less         # branch on less than

   slt    $at, $s3, $s4
   bne    $at, $zero, Less       # Example 4 on using $zero
```

- Register $at is reserved for use by the assembler
- Register $zero has an important role in simplifying the instruction set by offering useful variations

Home Exercise (25)

- Show how MIPS compilers use `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 that is always available by reading register **$zero** to create all relative conditions: equal (EQ), not equal (NQ), less than (LT), less than or equal (LE), greater than (GT), greater than or equal (GE)

- In other words, use MIPS instructions to implement the following pseudoinstructions:

```
blt  $s3, $s4, LT      # branch on less than
ble  $s3, $s4, LE      # branch on less than or equal
bgt  $s3, $s4, GT      # branch on greater than
bge  $s3, $s4, GE      # branch on greater than or equal
```

Group Exercise (26)

- **Decompile** the following MIPS assembly code to a C code segment

```
        slti  $at, $s5, 5
        beq   $at, $zero, Else
        add   $s6, $s5, $zero
        j     Exit
Else:   add   $s6, $zero, $zero
Exit:
```

- Assume that the decompiler associates registers $s5 and $s6 with integer variables i and x

Group Exercise (26): Answer

- Decompile the following MIPS assembly code to a C code segment
- Assume that the decompiler associates registers $s5 and $s6 with integer variables i and x

Assembly code with pseudoinstructions

`slti $at, $s5, 5` `beq $at, $zero, Else`	`bge $s5, 5, Else`
`add $s6, $s5, $zero`	`move $s6, $s5`
`j Exit`	`j Exit`
`Else: add $s6, $zero, $zero`	`Else: clear $s6`
`Exit:`	`Exit:`

```
if (i < 5)
    x = i;
else
    x = 0;
```

Individual Exercise (27)

- **Static code size** is the number of instructions a program has
- **Dynamic code size** is the actual number of instructions executed by the CPU for a specific program execution
- What is the static and dynamic code sizes of the MIPS assembly code segment in Exercise (26)
  - in number of instructions
  - Assume the two cases:
    - Register $s5 is initialized to 4
    - Register $s5 is initialized to 6

Individual Exercise (27): Answer

- What is the static and dynamic code sizes of the MIPS assembly code segment in Exercise (26)
  - in number of instructions
  - Static code size = 5 instructions

	Executed (Y/N)?	
	$s5 initialized to 4	$s5 initialized to 6
`slti $at, $s5, 5`	Y	Y
`beq $at, $zero, Else`	Y	Y
`add $s6, $s5, $zero`	Y	N
`j Exit`	Y	N
`Else: add $s6, $zero, $zero`	N	Y
`Exit:`		
Dynamic code size	**4 instructions**	**3 instructions**

Individual Exercise (28)

- What is the MIPS assembly code for the following machine instruction?

```
0x00 00 00 00
```

- Suggest an appropriate MIPS pseudoinstruction equivalent to that MIPS assembly code

Individual Exercise (28): Answer

- What is the MIPS assembly code for the following machine instruction?

```
0x00 00 00 00
```

```
sll  $zero, $zero, 0
```

  - Shift left logical register $zero by zero bits
  - Actually, this instruction does nothing

- Suggest an appropriate MIPS pseudoinstruction equivalent to that MIPS assembly code

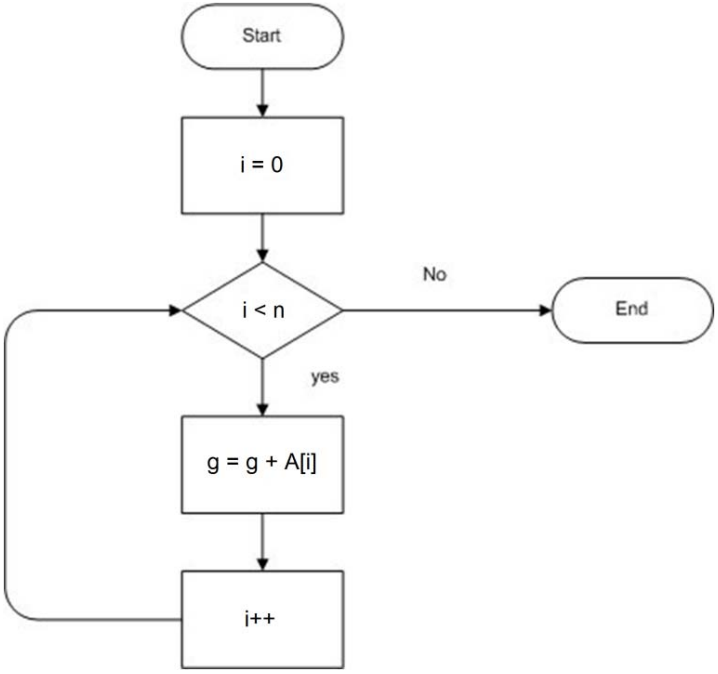
```
nop  # no operation
```

Individual Exercise (29)

- Assuming A is an integer array with base in $s4 and that the compiler associates the integer variables g, i, and n with the registers $s1, $s2, and $s3, respectively

- What is the compiled MIPS assembly code for the following C code segment?

```
for (i = 0; i < n; i++)
    g = g + A[i];
```



Individual Exercise (29): Answer

- Assuming A is an integer array with base in $s4 and that the compiler associates the integer variables g, i, and n with the registers $s1, $s2, and $s3, respectively
- What is the compiled MIPS assembly code for the following C code segment?

```
for (i = 0; i < n; i++)
    g = g + A[i];
```

```
        add  $s2, $zero, $zero  # i = 0
For:    slt  $t0, $s2, $s3      # test if i < n
        beq  $t0, $zero, Exit
        sll  $t1, $s2, 2        # $t1 = 4*i
        add  $t1, $t1, $s4      # $t1 has address of A[i]
        lw   $t2, 0 ($t1)       # $t2 = A[i]
        add  $s1, $s1, $t2      # g = g + A[i]
        addi $s2, $s2, 1        # increment i
        j    For
Exit:
```

Loop body

Addressing in Branches and Jumps

- **Design principle 3: Make the common case fast**
  - 16 bits for conditional branch addresses are far too small (maximum allowable program size will be 2^{16} bytes)
  - Conditional branches (`bne` and `beq`) are found in **loops** and **if statements**, so they tend to branch to a nearby instruction
  - About 50% of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away (**principle of locality!**)
- As the **Program Counter (PC)** holds the address of the instruction to be executed, we can branch within $\pm 2^{15}$ bytes of the instruction to be executed
- Use **PC-relative addressing**: add the value of the PC to the branch offset to get the jump-to address
- As it is convenient for the hardware to increment the PC early to point to the next instruction, branch offset is actually relative to the following instruction (PC + 4) as opposed to the current one (PC)
- Have the branch offset refer to the number of words to the destination (instead of bytes), which stretches the branch offset to be within $\pm 2^{15}$ words ($\pm 2^{17}$ bytes) of the current instruction

next PC (branch-to address) = (current PC + 4) + branch offset * 4

Addressing in Branches and Jumps (cont.)

- **Branch offset:** number of instructions counted from the **next instruction**

```
        beq   $t0, $t1, Skip
        nop   # 0 (start here)
        nop   # 1
        nop   # 2
Skip:   nop   # 3!
        ……
```

Offset = 3

```
Loop:   nop   # -5
        nop   # -4
        nop   # -3
        nop   # -2
        beq   $t0, $t1, Loop
        nop   # 0 (start here)
```

Offset = -5

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
I-format	op	rs	rt	16-bit immediate/address		

Fields of I-format branch instructions		
op	6 bits	Code for the comparison to perform first
rs	5 bits	1st argument register
rt	5 bits	2nd argument register
address	16 bits	Constant (**branch offset**) value embedded in the instruction

Note: 16-bit immediate/address: 1) **memory offset** (load/store), 2) **immediate** (e.g., addi), and 3) **branch offset** (beq/bne)

Individual Exercise (30)

- **Assemble** the `beq` MIPS instruction in the following code segment into real MIPS machine code:

```
        beq  $t0, $t1, L1
        nop
        nop
L1:     nop
```

  - Registers have numbers: `$t0 = 8, $t1 = 9`

Individual Exercise (30): Answer

```
        beq   $t0, $t1, L1
        nop   # 0 (start here)
        nop   # 1
L1:     nop   # 2 (branch offset = 2)
```

– Registers have numbers: `$t0 = 8, $t1 = 9`

op	rs	rt	immediate/address
4	8	9	2
000100	01000	01001	0000 0000 0000 0010

MIPS machine instruction: `00010001000010010000000000000010`

MIPS machine instruction (in hex): `0x1109 0002`

Addressing in Branches and Jumps (cont.)

- J-format instructions jump to addresses that have no reason to be near them
- J-format involves absolute (direct) addressing

- The 26-bit field in J-format instructions is a word address
  - it represents 28-bit byte addresses

- Since the PC is 32 bits, the address in J-format instructions replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged (**Pseudodirect addressing**)

- J-format instructions can only be used within memory blocks of a maximum size = 256 MB (28-bit address)

- The loader and linker must be careful to avoid placing a program across an address boundary of 256 MB, for otherwise a jump must be replaced by a jump register (`jr`) instruction preceded by other instructions to load the full 32-bit address into a register

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6

J-format	op	26-bit address

Fields of J-format instructions		
op	6 bits	Code for jump instructions j or jal
address	26 bits	Target address embedded in the instruction
Note: the jr instruction is an R-format instruction		

Individual Exercise (31)

- **Assemble** the `j` MIPS instruction in the following code segment into real MIPS machine code:

```
0x40000000          j     L1
…                   …
0x400000A4   L1:    nop
…                   …
```

Individual Exercise (31): Answer

- Calculating the address field of the j instruction
  1. Get address at L1 in hex: `0x400000A4`
  2. Drop the most significant hex digit: `0x 00000A4`
  3. Convert to binary: `00000000000000000000010100100`
  4. Drop the least significant two bits: `00000000000000000000101001`

op	Address

2	00000000000000000000101001

000010	00000000000000000000101001

MIPS machine instruction: `00001000000000000000000000101001`

MIPS machine instruction (in hex): `0x0800 0029`

Group Exercise (32)

- Assuming that the compiler associates integer variables i, j, k with registers $s3, $s4, and $s5 and the base of the integer array save is in $s6
- What is the compiled MIPS assembly code corresponding to the following C code segment? Use temporary registers $t0 and $t1, if needed

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

- If we assume that the loop is placed starting at location $80{,}000_{10}$ in memory, what is the real MIPS machine language version for this loop?

- Registers have numbers: `$s3 = 19, $s4 = 20, $s5 = 21, $s6 = 22, $t0 = 8, $t1 = 9`

Group Exercise (32): Answer

```
80000  Loop: sll $t1, $s3, 2      # $t1 = 4*i
80004        add $t1, $t1, $s6    # $t1 = address of save[i]
80008        lw  $t0, 0 ($t1)     # $t0 = save[i]
80012        bne $t0, $s5, Exit   # goto Exit if save[i] ≠ k
80016        add $s3, $s3, $s4    # i += j
80020        j   Loop             # goto Loop
80024  Exit:
```

Loop body

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	0	19	20	19	0	32
80020	2	20000				
80024						

Group Exercise (32): Answer (cont.)

000000	00000	10011	01001	00010	000000
000000	01001	10110	01001	00000	100000
100011	01001	01000	0000 0000 0000 0000		
000101	01000	10101	0000 0000 0000 0010		
000000	10011	10100	10011	00000	100000
000010	00 0000 0000 0100 1110 0010 0000				

Machine code: 00000000000100110100100010000000 = 0x00134880

Machine code: 00000001001101100100100000100000 = 0x01364820

Machine code: 10001101001010000000000000000000 = 0x8D280000

Machine code: 00010101000101010000000000000010 = 0x15150002

Machine code: 00000100111010010011000001000000 = 0x02749820

Machine code: 00001000000000000100111000100000 = 0x08004E20

MIPS machine instruction (in hex):

0x0013 4880 0136 4820 8D28 0000 1515 00020 0274 9820 0800 4E20

MIPS Addressing Modes Summary

- **Immediate addressing:** the operand is a constant within the instruction

```
addi $s1, $s2, 100
```

- **Register addressing:** the operand is a register

```
add  $s1, $s2, $s3
```

- **Base or displacement addressing:** the operand is at the memory location whose address is the sum of a register and a constant in the instruction

```
lw   $s1, 100 ($s2)
```

- **PC-relative addressing:** the address is the sum of the current PC and a constant (multiplied by 4) in the instruction

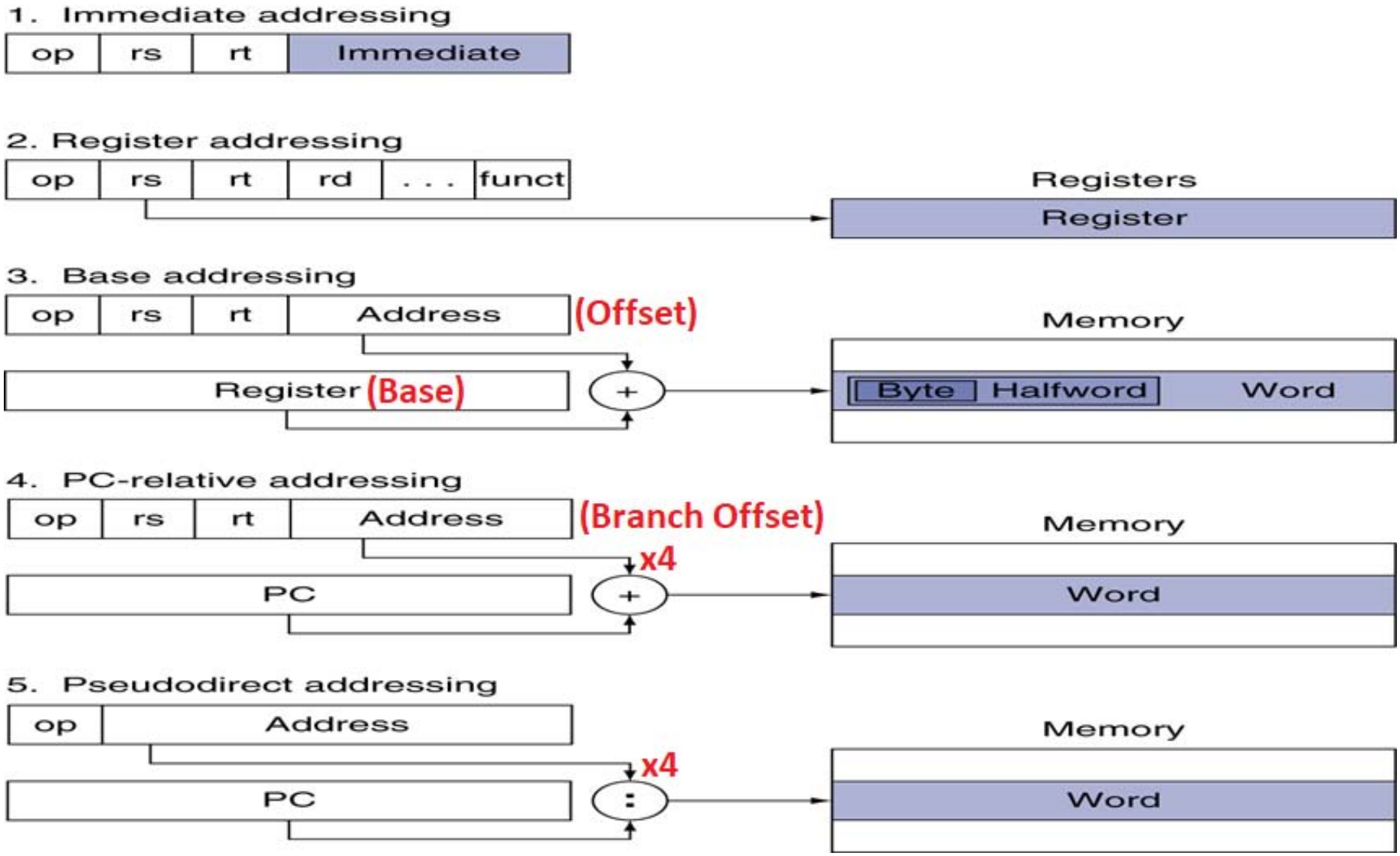
```
beq  $s1, $s2, 100
```

- **Pseudodirect addressing:** the jump address is a constant (26 bits) in the instruction (multiplied by 4) concatenated with the upper 4 bits of the PC

```
j    2500
```

- The **assembler** calculates the value to be stored in the **address field** of the machine code of an instruction while assembling a program
- The **processor** calculates **effective addresses** for branches, jumps, loads, and stores while executing an instruction

MIPS Addressing Modes Summary (cont.)



MIPS Addressing Modes Summary (cont.)

- **Relative** versus **absolute** (pseudodirect) addressing
  - **Branch instructions (beq, bne)**
    - Offset is **relative**
    - Next PC = (Current PC + 4) + Offset × 4
  - **Jump instructions (j, jal)**
    - Address is **absolute**
    - Next PC = (PC & 0xF0000000) | (Address × 4)
    - Absolute is relative to a 256 MB region of memory

MIPS Addressing Modes Summary (cont.)

- **Comparison between branch/jump instructions**

Conditional branches: beq, bne		
Offset:	16 bits	
Effective offset:	18 bits	Offset is × 4
Range:	2^{18}	PC ± 128 KB
Unconditional jumps: j, jal		
Address:	26 bits	
Effective address:	28 bits	Offset is × 4
Range:	2^{28}	Any address in current 256 MB block
Jump register: jr		
Address:	32 bits	Saved in a register
Range:	2^{32}	Any addressable memory location (4GB)

Individual Exercise (33)

- Given a branch on register $s0 being equal to register $s1

```
beq  $s0, $s1, L1
```

- The instruction is to branch far away, i.e., the branch offset is farther than can be represented in the 16-bit address field of the conditional branch instruction

- Show how the assembler might replace the above instruction by a pair of instructions to offer a much greater branching distance

Individual Exercise (33): Answer

- Given a branch on register $s0 being equal to register $s1

```
beq  $s0, $s1, L1
```

- The instruction is to branch far away, i.e., the branch offset is farther than can be represented in the 16-bit address field of the conditional branch instruction
- Show how the assembler might replace the above instruction by a pair of instructions to offer a much greater branching distance

- The assembler inserts an unconditional jump to the branch target and inverts the condition so that the branch decides whether to skip the jump or not

```
    beq  $s0, $s1, L1 ===>      bne $s0, $s1, L2
…                               j   L1
…                           L2:
…                           …
L1:                         L1:
```

- This will work as long as the program does not cross the 256 MB address boundary, otherwise a jr instruction should be used instead of the j one

Individual Exercise (34)

- **Assemble** the following MIPS instruction into real MIPS machine code:
  - `jr $ra`
  - Register has number: `$ra = 31`

Individual Exercise (34): Answer

- Assemble the following MIPS instruction into real MIPS machine code:
  - `jr $ra`
  - Register has number: `$ra = 31`

op	rs	rt	rd	shamt	funct
0	31	0	0	0	8
000000	11111	00000	00000	00000	001000

Machine code: `00000011111000000000000000001000 = 0x03E00008`

  - The encoding of `jr` is 0 in the **op** field and 8 in the **funct** field
  - The **rt**, **rd**, and **shamt** fields are unused, and thus are set to 0

Home Exercise (35)

- "**Make the common case fast**" motivates many features of the MIPS instruction set

- Give two examples supporting this sentence!

Supporting Procedures

- **Procedure calls proceed in the following six steps:**
  1. The calling program (caller) puts the parameter values in **$a0-$a3**
  2. The caller uses `jal X` to jump to procedure X (callee)
  3. The callee acquires the storage resources it needs
  4. The callee performs its task
  5. The callee places the results in **$v0-$v1**
  6. Finally, the callee returns control to the caller at the point of origin using `jr` **$ra**

- **MIPS subroutine call:** jump-and-link (`jal`) jumps unconditionally to a procedure address and simultaneously saves the address of the following instruction (PC + 4) in **$ra**

```
jal Procedure  # Procedure is a word address (J-format)
jr  $ra        # unconditional jump to the address in $ra
```

Supporting Procedures (cont.)

Name	Register number (decimal)	Usage	Preserve on call?
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	**2-3**	**procedure return values and expression evaluation**	**no**
$a0-$a3	**4-7**	**procedure arguments (parameters)**	**no**
$t0-$t7	8-15	temporary registers	no
$s0-$s7	16-23	general purpose saved registers	yes
$t8-$t9	24-25	more temporary registers	no
$k0-$k1	26-27	reserved for the OS	n.a.
$gp	**28**	**global pointer**	**yes**
$sp	**29**	**stack pointer**	**yes**
$fp	**30**	**frame pointer**	**yes**
$ra	**31**	**procedure return address**	**yes**

Supporting Procedures (cont.)

- **Using a stack (last-in first-out or LIFO) with procedures:**
  - **Stack:** a linear list for which all insertions (**push** operations) and deletions (**pop** operations) are performed at one end called stack **top**
  - The **stack pointer** (**$sp**) points to the most recently allocated address
  - Grows from **higher** addresses to **lower** addresses
  - Ideal structure for **spilling registers**, **extra arguments** (**more than 4**), and **extra return values** (**more than 2**)
  - Used to save **local variables** that do not fit in registers

- **Preserving registers** (saving on the stack):
  - **$s0-$s7**: 8 saved registers that are preserved on a procedure call (the callee **saves used ones** on the stack and restores them upon return)
  - **$t0-$t9**: 10 temporary registers that are not preserved by the callee
  - **$a0-$a3**: 4 procedure arguments (parameters) registers that are not preserved by the callee
  - **$v0-$v1**: 2 procedure return registers that are not preserved by the callee
  - **$ra**: procedure return address register that is preserved on a procedure call

Supporting Procedures (cont.)

- **Types of procedures:**
  - **Main procedure:** call other procedures
    - **required (as caller) to preserve:** any **temporary registers** (**$t0-$t9**), **argument registers** (**$a0-$a3**),or **return registers** (**$v0-$v1**) needed after the call
    - **require its callee to preserve:** only **saved registers** (**$s0-$s7**) used by the callee
  - **Leaf procedures:** do not call other procedures
    - **required (as callee) to preserve:** only **saved registers** (**$s0-$s7**) used by itself (the callee)
  - **Nested procedures:** call other procedures
    - **required (as caller) to preserve:** any **temporary registers** (**$t0-$t9**), **argument registers** (**$a0-$a3**),or **return registers** (**$v0-$v1**) needed after the call, and **return address register** (**$ra**)
    - **required (as callee) to preserve:** only **saved registers** (**$s0-$s7**) used by itself (the callee)
    - **require its callee to preserve:** only **saved registers** (**$s0-$s7**) used by the callee

Group Exercise (36)

- What is the compiled MIPS assembly code for the following C procedure?

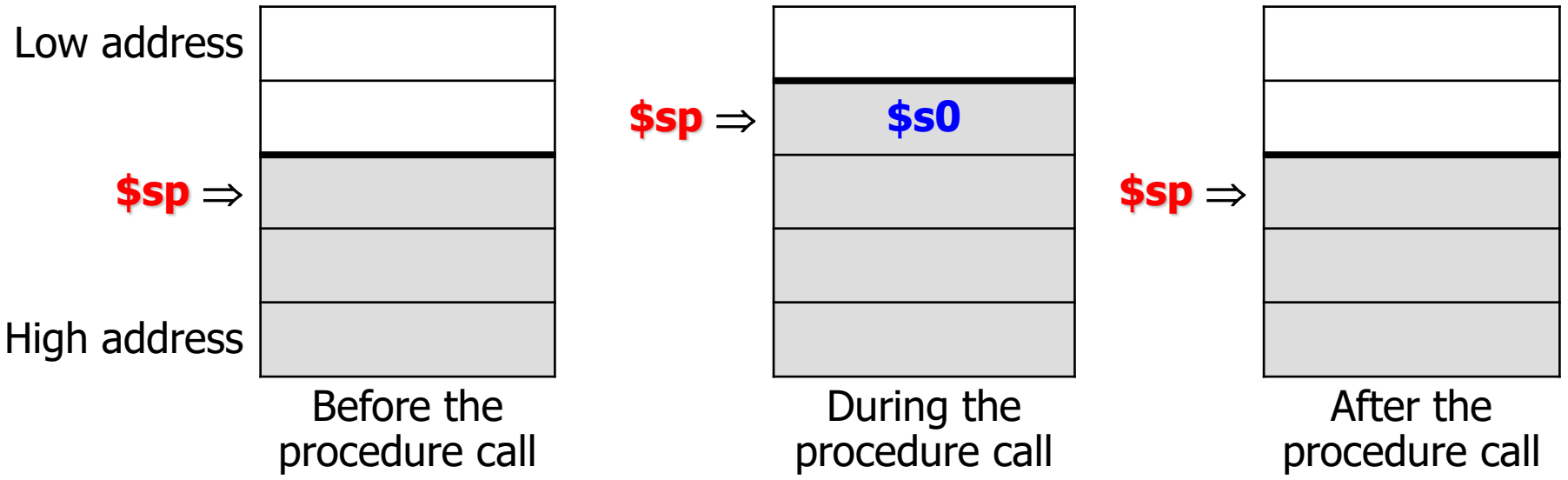
```
int leaf_ex (int g, int h, int i, int j) {
   int f;

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

- Parameter integer variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and integer variable f corresponds to $s0

- Trace the stack contents (stack image) before, during, and after the procedure call

Group Exercise (36): Answer

```
leaf_ex:  addi  $sp, $sp, -4      # make 1 space on stack
          sw    $s0, 0 ($sp)      # save $s0 on stack
          add   $t0, $a0, $a1     # $t0 = g + h
          add   $t1, $a2, $a3     # $t1 = i + j
          sub   $s0, $t0, $t1     # f = (g + h) - (i + j)
          add   $v0, $s0, $zero   # $v0 = $s0 to return
          lw    $s0, 0 ($sp)      # restore $s0
          addi  $sp, $sp, 4       # adjust $sp
          jr    $ra               # return back to caller
```



Group Exercise (37)

- What is the compiled MIPS assembly code for the following C program?

```
main() {
   int a, b, c;

   c = sum(a,b);
   …
}

int sum (int x, int y) {
   return (x + y);
}
```

- Parameter integer variables x and y correspond to the argument registers $a0 and $a1, and integer variables a, b, and c correspond to $s0, $s1, and $s2

Group Exercise (37): Answer

```
main() {
   int a, b, c;

   c = sum(a,b);
   …
}

int sum (int x, int y) {
   return (x + y);
}
```

```
main:   add   $a0, $s0, $zero      # x = a
        add   $a1, $s1, $zero      # y = b
        jal   sum                  # $ra = return address
        add   $s2, $v0, $zero      # c = sum(a,b)
        ...

sum:    add   $v0, $a0, $a1
        jr    $ra
```

Group Exercise (38)

- Consider the following C-language program
- Integer variables c, d, and i are stored in registers $s0, $t1, and $s0, respectively

```
main(){
   int i = calculate(1,2);
   …
   …
}

int calculate(int x, int y){
   return(evaluate(x+y,3));
}
```

```
int evaluate(int a, int b){
   int c = square(a) + b;
   return c;
}

int square (int u){
   int d = u * u;
   return d;
}
```

- Indicate if functions main(), calculate(), evaluate(), and square() need to preserve registers $a0, $a1, $s0, $t1, and $ra or not. Justify!

Group Exercise (38): Answer

- Integer variables c, d, and i are stored in registers $s0, $t1, and $s0, respectively

```
main(){
   int i = calculate(1,2);
   …
   …
}

int calculate(int x, int y){
   return(evaluate(x+y,3));
}
```

```
int evaluate(int a, int b){
   int c = square(a) + b;
   return c;
}

int square (int u){
   int d = u * u;
   return d;
}
```

Functions	Registers	Need to preserve?	Justification	Comment
main()	$a0	No	Not needed by the function after the call	
	$a1	No	Not needed by the function after the call	
	$s0	No	Callee responsibility to preserve	Holds local variable i
	$t1	No	Not used by the function	
	$ra	No	Callee responsibility to preserve	
calculate()	$a0	No	Not needed by the function after the call	
	$a1	No	Not needed by the function after the call	
	$s0	No	Not used by the function	
	$t1	No	Not used by the function	
	$ra	Yes	It modifies it and it is its responsibility to preserve	

Group Exercise (38): Answer (cont.)

- Integer variables c, d, and i are stored in registers $s0, $t1, and $s0, respectively

```
main(){
   int i = calculate(1,2);
   …
   …
}

int calculate(int x, int y){
   return(evaluate(x+y,3));
}
```

```
int evaluate(int a, int b){
   int c = square(a) + b;
   return c;
}

int square (int u){
   int d = u * u;
   return d;
}
```

Functions	Registers	Need to preserve?	Justification	Comment
evaluate()	$a0	No	Not needed by the function after the call	
	$a1	Yes	Needed by the function after the call	
	$s0	Yes	It modifies it and it is its responsibility to preserve	Holds local variable c (not related to i)
	$t1	No	Not used by the function	
	$ra	Yes	It modifies it and it is its responsibility to preserve	
square()	$a0	No	Not modified by the function	
	$a1	No	Not used by the function	
	$s0	No	Not used by the function	
	$t1	No	It modifies it but it is not its responsibility to preserve	
	$ra	No	Not modified by the function	

Group Exercise (39)

- What is the compiled MIPS assembly code for the following C program?

```
main(){
   int a=5,b=3,c;

   c = sum2(a,b);
   …
}
```

```
int sum2(int x, int y){

   return(sum(x,x) + y);
}
```

```
int sum(int p, int q){
   int m;

   m = p + q;
   return(m);
}
```

- Integer variables a, b, and c of the main() function correspond to $s0, $s1, and $s2
- Integer variable m of the sum() function corresponds to $s0
- Assume the functions main(), sum2(), and sum() are placed starting at locations 1000_{10}, 2000_{10}, and 4000_{10}, respectively in memory
- Assume that the stack pointer has an initial value of 8000_{10}
- Trace the stack contents after the execution of each instruction
- Trace the contents of the registers $s0, $s1, $s2, $ra, $a0, $a1, $v0, and $sp after the execution of each instruction

Group Exercise (39): Answer

```
main(){
   int a = 5, b = 3, c;

   c = sum2(a,b);
   …
}
```

```
int sum2(int x, int y){

   return(sum(x,x) + y);
}
```

```
int sum(int p, int q){
   int m;

   m = p + q;
   return(m);
}
```

```
1000  main:  add   $a0, $s0, $zero
1004         add   $a1, $s1, $zero
1008         jal   sum2
1012         add   $s2, $v0, $zero
             …
```

```
2000  sum2:  addi  $sp, $sp, -8
2004         sw    $ra, 4 ($sp)
2008         sw    $a1, 0 ($sp)
2012         add   $a1, $a0, $zero
2016         jal   sum
2020         lw    $a1, 0 ($sp)
2024         lw    $ra, 4 ($sp)
2028         addi  $sp, $sp, 8
2032         add   $v0, $v0, $a1
2036         jr    $ra
```

```
4000  sum:   addi  $sp, $sp, -4
4004         sw    $s0, 0 ($sp)
4008         add   $s0, $a0, $a1
4012         add   $v0, $s0, $zero
4016         lw    $s0, 0 ($sp)
4020         addi  $sp, $sp, 4
4024         jr    $ra
```

```
1000    main:   add  $a0,$s0,$zero
1004            add  $a1,$s1,$zero
1008            jal    sum2
1012            add  $s2,$v0,$zero
                …
2000    sum2:   addi $sp, $sp, -8
2004            sw $ra, 4($sp)
2008            sw $a1, 0($sp)
2012            add $a1,$a0,$zero
2016            jal   sum
2020            lw $a1, 0($sp)
2024            lw $ra, 4($sp)
2028            addi $sp, $sp, 8
2032            add $v0, $v0, $a1
2036            jr $ra
                …
4000    sum:    addi $sp, $sp, -4
4004            sw $s0, 0($sp)
4008            add $s0, $a0, $a1
4012            add $v0,$s0,$zero
4016            lw $s0, 0($sp)
4020            addi $sp, $sp, 4
4024            jr $ra
```

Inst. Execution Order	$s0	$s1	$s2	$ra	$a0	$a1	$v0	$sp
1000	5	3	?	?	5	?	?	8000
1004	5	3	?	?	5	3	?	8000
1008	5	3	?	1012	5	3	?	8000
2000	5	3	?	1012	5	3	?	7992
2004	5	3	?	1012	5	3	?	7992
2008	5	3	?	1012	5	3	?	7992
2012	5	3	?	1012	5	5	?	7992
2016	5	3	?	2020	5	5	?	7992
4000	5	3	?	2020	5	5	?	7988
4004	5	3	?	2020	5	5	?	7988
4008	10	3	?	2020	5	5	?	7988
4012	10	3	?	2020	5	5	10	7988
4016	5	3	?	2020	5	5	10	7988
4020	5	3	?	2020	5	5	10	7992
4024	5	3	?	2020	5	5	10	7992
2020	5	3	?	2020	5	3	10	7992
2024	5	3	?	1012	5	3	10	7992
2028	5	3	?	1012	5	3	10	8000
2032	5	3	?	1012	5	3	13	8000
2036	5	3	?	1012	5	3	13	8000
1012	5	3	13	1012	5	3	13	8000

Stack

	address	data
	7984	
$sp →	7988	5
$sp →	7992	3
	7996	1012
$sp →	8000	?
	8004	?
	8008	?

Slide show (click here)

Home Exercise (40)

- Show how the stack pointer ($sp) register is preserved on a procedure call

Home Exercise (41)

- List hardware features through which MIPS supports procedures

Home Exercise (42)

- List possible overheads of organizing a piece of assembly code into a set of procedures

CISC versus RISC

- **CISC versus RISC philosophy: should instruction set be**
  - complicated or simple?
  - rich or fundamental in operations?
  - sophisticated or basic in addressing modes?
  - implemented in hardware (microprogramming) or in software/compiler?

- **Instruction set design:**
  - **CISC:** complex hardware drives instruction set choices
  - **RISC:** compiler technology drives instruction set choices

- **Software:** levels of machine languages in relation to that of the high-level languages

- **Hardware:** tradeoffs made in the utilization of hardware resources

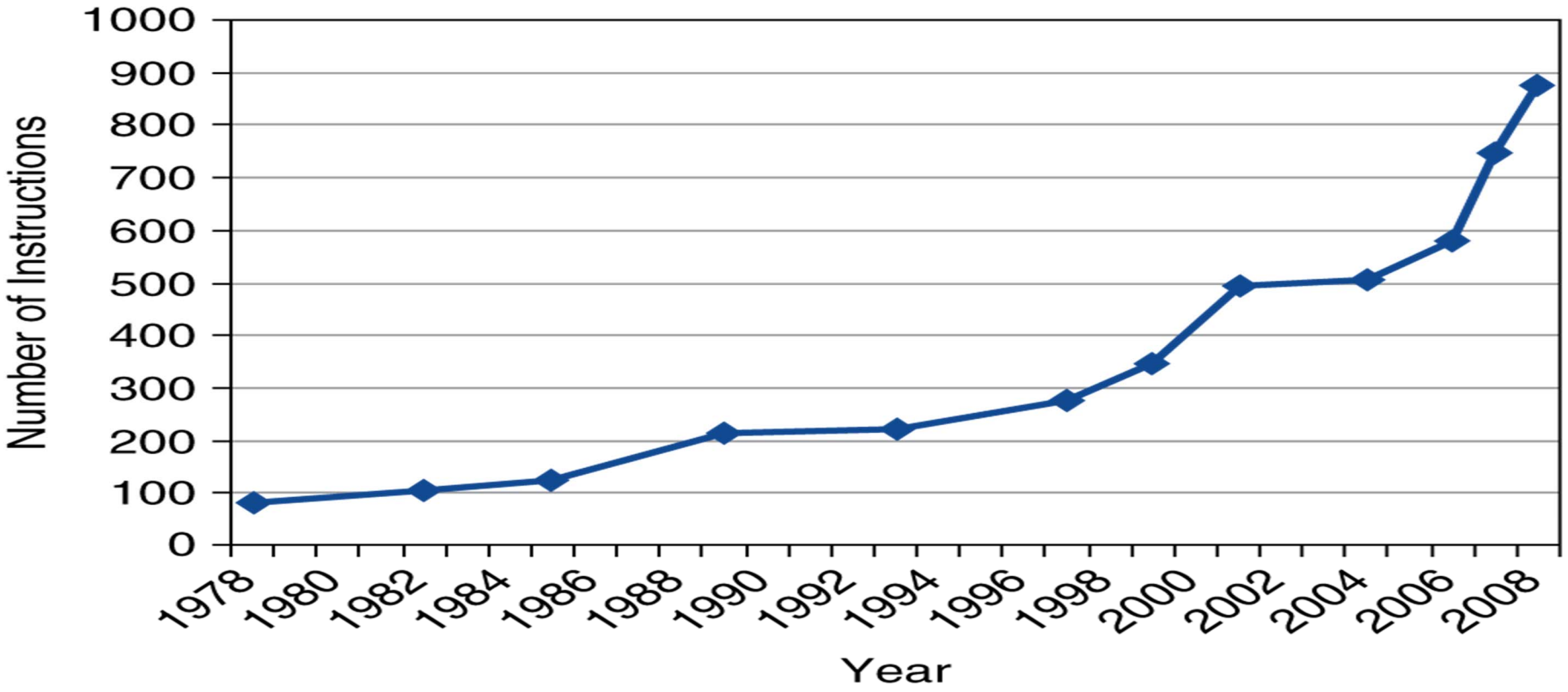
- **Hardware/software (runtime/compile-time) tradeoffs:** where to put functionality

Development of CISC

- To design **faster machines**, CISC moves functions from software into hardware and microcode
  - **Microprogramming** encourages complex designs by making it easy to change the control algorithm for complex instructions or add new ones
  - added architectural features have adverse effects on implementation
- To reduce the **Semantic gap** between HLLs and assembly languages, CISC introduces more complex instructions of high semantic level
  - resulted in a mismatch between both levels
- New models have **backward-compatibility** with existing same family ones
  - leads to supersetting and feature proliferation
- CISC follows an **evolutionary path** and develops machines that are extensions of previous generations
  - led to inefficient designs
- **CAD tools** aid designers in handling the inherent complexity of large and complex architectures
  - automatically designed hardware is not necessarily optimized

CISC Case Study: x86 ISA

- The x86 architecture has grown dramatically
- The average is more than one instruction per month over its 30-year lifetime!
- Less is more!



Development of RISC

- CISC goal was to reduce the number of instructions executed by a program
  - this reduction can occur at the cost of simplicity
  - this reduction can increase the time a program takes to execute because the instructions are slower
  - this slowness maybe the result of:
    - a slower clock cycle time or
    - requiring more clock cycles than a simpler sequence for execution

- RISC has moved toward simpler instructions to avoid CISC problems

- **Motivations:**
  - Improvements in programming languages
  - Improvements in compiler technology
  - Improvements in memory cost

- This led to less programming being done at the assembly level

Development of RISC: Instruction Set Studies

- **Operations performed:** relative frequency of operations
  - determine the required operational units
- **Operands used:** type, size, and structure
  - determine the storage organization and addressing mode
- **Execution sequencing:** control transfers
  - determine the control and pipeline organization
- **Critical loops:** program portions, where most execution time is spent
  - determine bottlenecks and dedicated hardware support desired
- **Static measurement:** size of program
  - determine memory requirement and instruction fetch techniques
- **Dynamic measurement:** executed number of instructions
  - determine dynamic program behavior at runtime

Development of RISC: Instruction Set Statistics

- The simplest **operations** are the ones that are executed most of the time
- Many **assignment** statements are especially of the type **A = B;**
- Programs rarely use more than **16** general registers
- **Register allocation** can be done in software and/or hardware
- **if-statements** (branches) represent a high percentage of statements
- **Procedures** are called with a high frequency and are costly in terms of execution time
- More than 90% of procedure calls has less than 6 arguments passed
- Instruction decoding time is minimized if instruction length equals machine word size

- Instruction sets could be measured by how well compilers use them as opposed to how well assembly language programmers use them

- **Fact: Programmers and compilers use only a favorite instruction subset even if the instruction set is large and rich**

Common RISC Traits

- Operations are reduced
- Addressing modes are reduced
- Instructions are of a fixed length
- Instruction formats are simple and do not cross word boundaries
- **RISCs are load-store architectures:**
  - Operations are register-to-register, with only LOADs and STOREs accessing memory

RISC Advantages Claimed

- A simple instruction set is a better match for a compiler and especially a best target for optimizing compiler technology
- More compile-time effort offers an opportunity to explicitly move static runtime complexity into the compiler
- Low overhead per instruction makes single-cycle possible
- Faster instruction decoding and execution make single-cycle instruction possible
- Hardwired control eliminates runtime translation
  - reduces hardware and improves performance
  - provides the fastest possible single-cycle operation
- Freed hardware space can be devoted to innovative uses (e.g., caches)
- **Simpler design requires less design and debug time!**

RISC Disadvantages Realized

- Small and simple instruction set makes programs relatively lengthy
- Lengthy programs needs
  - more secondary storage
  - bigger cache to avoid memory contention
  - higher memory bandwidth
  - a larger working set for temporal and spatial locality
- System software (compiler and operating system) needed are more complex
- Needs pipeline scheduling to reduce hazards and stalls (as will be shown later!)

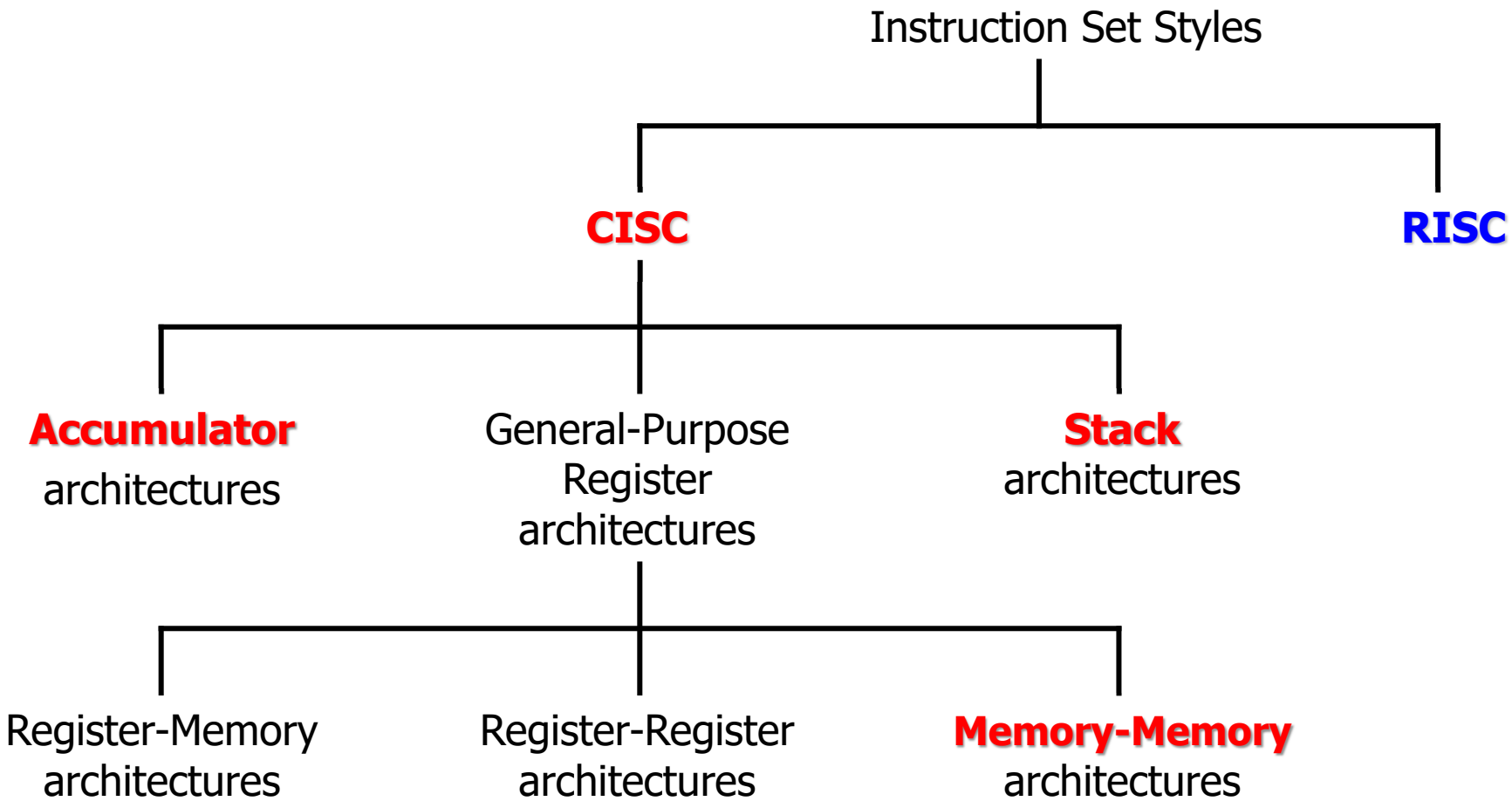
Home Exercise (43)

- Explain how RISC bridges the semantic gap between HLLs and assembly languages

Home Exercise (44)

- In a table, differentiate between RISC and CISC with respect to each of the following aspects:
  1. registers,
  2. data types,
  3. memory access,
  4. addressing modes,
  5. instruction set size,
  6. instruction set complexity,
  7. instruction length,
  8. instruction format,
  9. required compiler design,
  10. static program size,
  11. control unit implementation, and
  12. driving technology.

Instruction Set Styles



Accumulator Architectures

- Old computers had a single register for arithmetic and logical instructions
- **Accumulator:** a register where all operations would accumulate in
- The accumulator holds both the operation source and destination
- The final operand is found in memory; no registers are specified:

```
add 200     # Acc = Acc + Memory [address 200]
```

- All variables are allocated to memory in accumulator machines
- In this style of machines, variables are always spilled to memory
- It takes many more instructions to execute a program

- This type of machines is classified as: **1-address architectures**
  - To know the typical number of memory or register addresses per instruction, an instruction that takes two operands and produces a result (e.g., add) is considered

- Next generation computers added dedicated registers (special-purpose registers, extended accumulators) for specific operations; array reference indices, multiply and division separate accumulators, stack-top pointer, etc.
- This type of machines allowed also registers to hold operands
- 8086 is an accumulator architecture!

Individual Exercise (45)

- What is the accumulator-style assembly code for the following C code?

```
A = B + C;
B = A - C;
```

- A, B, and C are integer variables in memory

Individual Exercise (45): Answer

- What is the accumulator-style assembly code for the following C code?

```
A = B + C;
B = A – C;
```

- A, B, and C are integer variables in memory

```
load   AddressB     # Acc = Memory [AddressB]
add    AddressC     # Acc = B + Memory [AddressC]
store  AddressA     # Memory [AddressA] = Acc
sub    AddressC     # Acc = A - Memory [AddressC]
store  AddressB     # Memory [AddressB] = Acc
```

General-Purpose Register Architectures

- The generalization of the dedicated-register machine allows all the registers to be used for any purpose
- This type of machines is classified as: **3-address architectures**
- **There are three kinds of these architectures:**
  - those that allow one operand to be in memory as found in accumulator machines, called **register-memory** architectures (e.g., IBM 360 and 80386)
  - those that demand all three operands to be in registers, called either **load-store** or **register-register** architectures (e.g., CDC)
  - those that allow all three operands of each instruction to be in memory, called **memory-memory** architectures (e.g., VAX)

Individual Exercise (46)

- What is the memory-memory style assembly code for this C code?

```
A = B + C;
B = A - C;
```

- A, B, and C are integer variables in memory

Individual Exercise (46): Answer

- What is the memory-memory style assembly code for this C code?

```
A = B + C;
B = A - C;
```

- A, B, and C are integer variables in memory

```
add   AddressA, AddressB, AddressC
sub   AddressB, AddressA, AddressC
```

Compact Code Architectures

- **It is important to keep programs small:**
  - to minimize memory usage
  - to reduce number of instructions executed
  - to minimize transmit time on the Internet for network computers
- **Techniques to generate a small size code:**
  - **use variable length instructions**
    - **Advantages:**
      - match the varying operand specifications,
      - minimize code size, and
      - make assembly programming easier
    - **Examples:**
      - Intel 8086 instructions are from 1 to 17 bytes long
      - IBM 360 instructions are 2, 4, or 6 bytes long
      - VAX instruction lengths are anywhere from 1 to 54 bytes
  - **use a stack model of execution**
    - **Example:** old HP calculators
- Java has a compact code architecture that is based on variable length instructions and a stack architecture

Stack Architectures

- In stack architectures, all operations occur on top of the stack
- Operations remove their operands from the stack and replace them with the result
- Operands are pushed on the stack from memory or popped off the stack into memory
- This type of machines is classified as: **0-address architectures**

- **Advantages:**
  - simplify compilers by eliminating register allocation
  - easy management of temporary variables during computation
  - fewer operand fetches
  - faster program access and less memory contention
  - produce compact instruction encoding (only opcode is required)
- **Disadvantages:**
  - concurrent execution of instructions is difficult to manage
  - it is hard to reuse data that has been fetched or calculated without repeatedly going to memory

Individual Exercise (47)

- What is the stack-style assembly code for this C code?

```
A = B + C;
B = A - C;
```

- A, B, and C are integer variables in memory

Individual Exercise (47): Answer

- What is the stack-style assembly code for this C code?

```
A = B + C;
B = A - C;
```

- A, B, and C are integer variables in memory

```
push  AddressB
push  AddressC
add
dup                 # duplicate stack top (A)
pop   AddressA
push  AddressC
sub
pop   AddressB
```

Individual Exercise (48)

- Sometimes architectures are characterized according to the typical number of memory addresses per instruction
- Commonly used terms are 0, 1, 2, and 3 addresses per instruction
- Associate the following different architectural styles
  - Accumulator
  - Memory-memory
  - Load-store
  - Stack

  with each category

Individual Exercise (48): Answer

- Sometimes architectures are characterized according to the typical number of memory addresses per instruction
- Commonly used terms are 0, 1, 2, and 3 addresses per instruction
- Associate the following different architectural styles
  - Accumulator
  - Memory-memory
  - Load-store
  - Stack

  with each category

- To know the typical number of memory addresses per instruction, the nature of a typical instruction must be agreed upon
- For the purpose of categorizing computers as 0-, 1-, 2-, 3-address computers, an instruction that takes two operands and produces a result, for example, add, is traditionally taken as typical

Individual Exercise (48): Answer (cont.)

- **Accumulator:**
  - An add on this architecture reads one operand from memory, one from the accumulator, and writes the result in the accumulator
  - Only the location of the operand in memory needs to be specified by the instruction
  - **Category: 1-address architecture**

- **Memory-memory:**
  - Both operands are read from memory and the result is written to memory
  - All locations must be specified
  - **Category: 3-address architecture**

Individual Exercise (48): Answer (cont.)

- **Load-store:**
  - Both operands are read from registers and the result is written to a register (just like memory-memory)
  - All locations must be specified; however, location addresses are much smaller—5 bits for a location in a typical register file versus 32 bits for a location in a common memory
  - **Category: 3-address architecture**

- **Stack:**
  - Both operands are read (removed) from the stack (top of stack and next to top of stack), and the result is written to the stack (at the new top of stack)
  - All locations are known; none need be specified
  - **Category: 0-address architecture**

Home Exercise (49)

- Give an example of a 2-address architecture

Fallacies and Pitfalls

- **Fallacy:** more powerful instructions mean higher performance
- **Fallacy:** write in assembly language to obtain the highest performance
- **Fallacy:** the importance of commercial binary compatibility means successful instruction sets do not change
- **Pitfall:** forgetting that sequential word addresses in machines with byte addressing do not differ by one!
- **Pitfall:** using a pointer to an automatic variable outside its defining procedure