CCSYA

Instructions: Language of the Computer

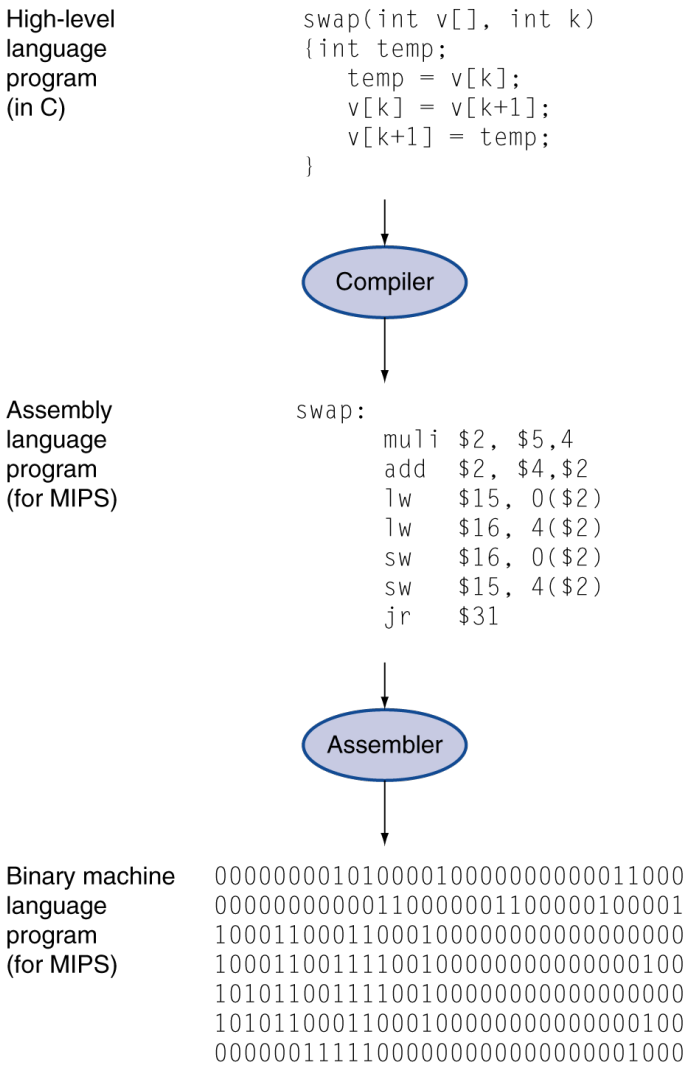
Part I

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Levels of Program Code

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data



Instruction Set

- **To command a computer's hardware, you must speak its language**
- **The words of a computer's language are called *instructions*, and its vocabulary is called an instruction set**
- **Computer languages are quite similar, more like regional dialects than like independent languages**
  - Hence, once you learn one, it is easy to pick up others

Complex Instruction Set (CISC)

- **Primary goal is to complete a task in few lines of assembly instruction as possible**
  - Each instruction can execute several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction
- **Complexity is on the hardware**
  - Variable length instructions
  - Multiple addressing modes
  - Instructions require multiple clock cycles to execute
  - …
- **VAX, Motorola 68000, Intel x86, …**

The Intel x86 ISA

- **Evolution with backward compatibility**
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

The Intel x86 ISA

- **Further evolution…**
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

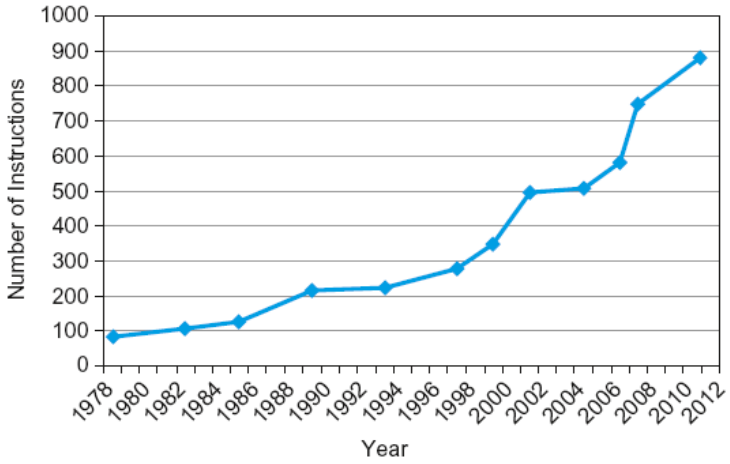
The Intel x86 ISA

- **And further…**
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions

- **If Intel didn't extend with compatibility, its competitors would!**
  - Technical elegance ≠ market success

Fallacies

- **Powerful instruction ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- **Backward compatibility ⇒ instruction set doesn't change**
  - But they do accrete more instructions



x86 instruction set

Implementing x86 ISA

- **Complex instruction set makes implementation difficult**
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Micro engine similar to RISC
  - Market share makes this economically viable
- **Comparable performance to RISC**
  - Compilers avoid complex instructions

x86 ISA

- **What the x86 lacks in style, it made up for in market size, making it beautiful from the right perspective**
  - Its saving grace is that the most frequently used x86 architectural components are not too difficult to implement

- **In the PostPC era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device**

Reduced Instruction Set (RISC)

- **A small set of simple and general instructions that can be executed within one clock cycle**
- **Complexity is on the software**
  - Primary goal is to speedup individual instruction execution at the cost of a higher number of instructions in a program
  - A larger number of registers
- **Load/store architecture**
  - Memory is accessed through specific instructions rather than as a part of most instructions
- **ARM, MIPS, IBM PowerPC, Sun Sparc, …**

The MIPS Instruction Set

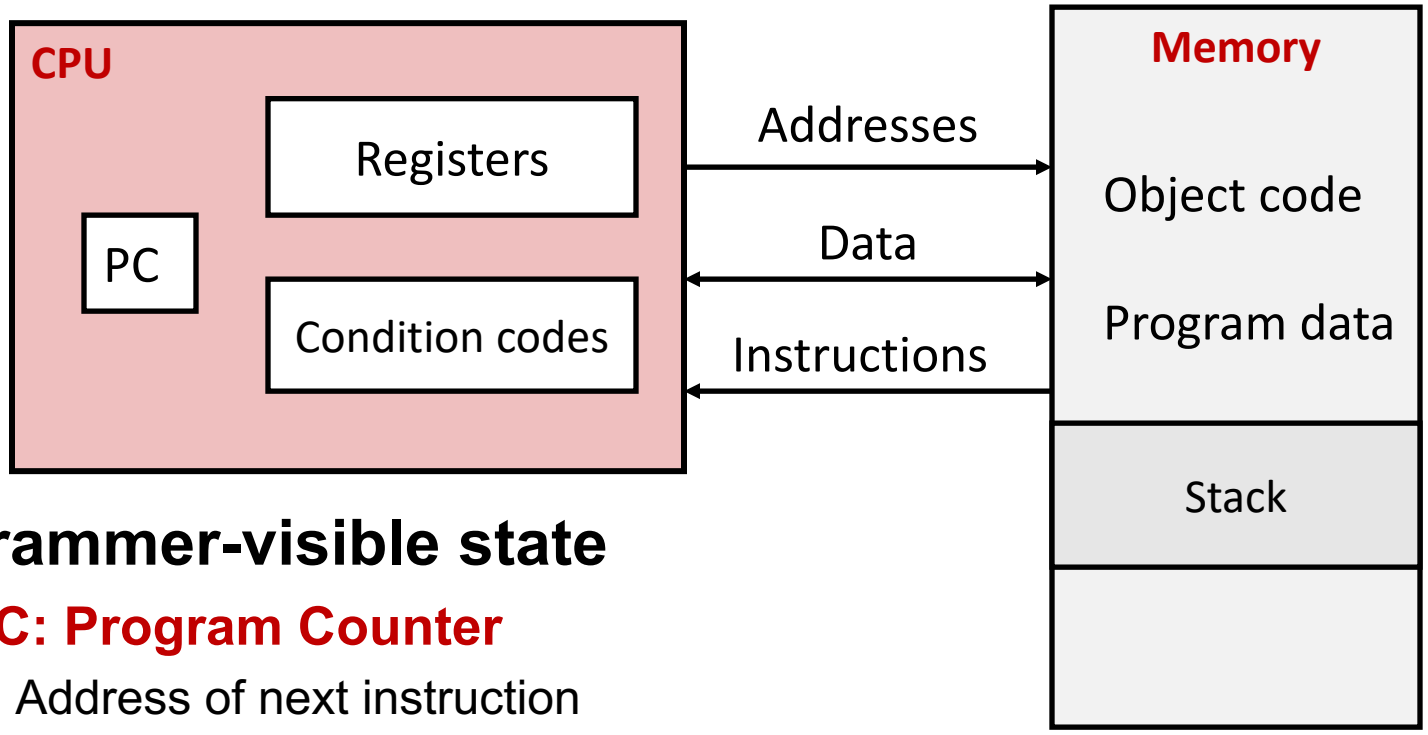
- **Used as the example throughout the course**
- **Stanford MIPS commercialized by MIPS Technologies (www.mips.com)**
- **Similar ISAs have a large share of embedded core market**
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - ARMv8 is very similar to MIPS

MIPS Simplified Implementation

- **We will consider a simplified implementation of the MIPS instruction set architecture**
  - As a way to illustrate how it determines many aspects of the implementation, and how the choice of different implementation strategies affects the clock rate and CPI for the computer

- **A subset of the core MIPS instruction set:**
  - Arithmetic-logical instructions: add, sub, and, or, slt, …
  - Memory reference instructions: lw(load word), sw(store word)
  - Control transfer instructions: beq(branch equal), bne (branch not equal), j(jump unconditionally), …

Assembly programmer's view



Programmer-visible state

- **PC: Program Counter**
  - Address of next instruction
- **Registers**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

Arithmetic Operations

- **Example: add**
  - Two sources and one destination

```
add a, b, c        # a gets b + c
```

- **All arithmetic operations have this form**

Design Principle 1: Simplicity favors regularity

Regularity makes implementation simpler

Simplicity enables higher performance at lower cost

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”
- **Assembler names**
  - `$t0`, `$t1`, …, `$t9` for temporary values
  - `$s0`, `$s1`, …, `$s7` for saved variables

Design Principle 2: Smaller is faster

The designer must balance the craving of programs for more registers with the designer’s desire to keep the clock cycle fast

Register Operand Example

- **C code:**

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

- **Compiled MIPS code:**
  - f, …, j in `$s0`, …, `$s4`

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

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
```

Design Principle 3: Make the common case fast

Small constants are common
Immediate operand avoids a load instruction
Operations are much faster and use less energy than if constants were loaded from memory

The Constant Zero

- **MIPS register 0 ($zero) is the constant 0**
  - Cannot be overwritten
  - Its main role is to simplify the instruction set by offering useful variations

- **Useful for common operations**
  - E.g., move between registers

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

Bitwise Operations

- **Instructions for bitwise manipulation**
  - Useful for extracting and inserting groups of bits in a word

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

Shift Operations

- **Shift left logical**
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i

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

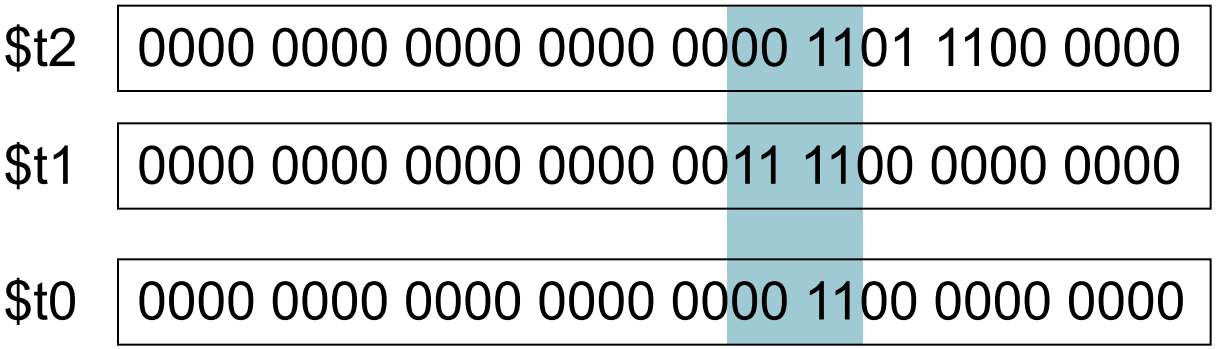
- **Shift right logical**
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

```
srl $t2,$s0,4      # $t2 = $s0 >> 4
```

AND Operations

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

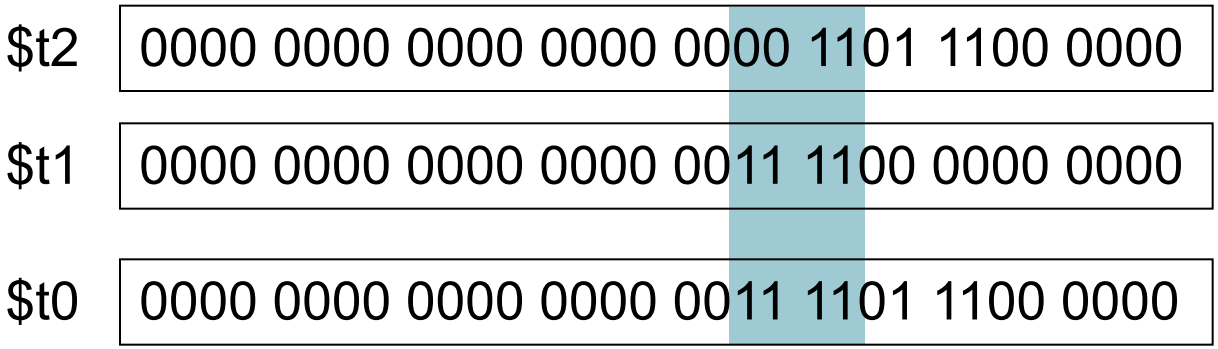
```
and $t0, $t1, $t2
```



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

Arithmetic and Bitwise Instructions

`add $1, $2, $3`	$s1 = $2 + $3	add
`addu $1, $2, $3`	$s1 = $2 + $3	add unsigned
`addi $1, $2, imm`	$s1 = $2 + imm	add immediate
`addiu $1, $2, imm`	$s1 = $2 + imm	add immediate unsigned
`sub $1, $2, $3`	$s1 = $2 - $3	subtract
`and $s1, $s2, $s3`	$s1= $s2 & $s3	bitwise and
`andi $s1, $s2, imm`	$s1= $s2 & $s3	bitwise and immediate
`or $s1, $s2, $s3`	$s1= $s2 | $s3	bitwise or
`nor $s1, $s2, $s3`	$s1= ~($s2|$s3)	bitwise not
`sll $s1, $s2, imm`	$s1= $s2 << imm	shift left logical
`srl $s1, $s2, imm`	$s1= $s2 >> imm	shift right logical

Registers vs. Memory

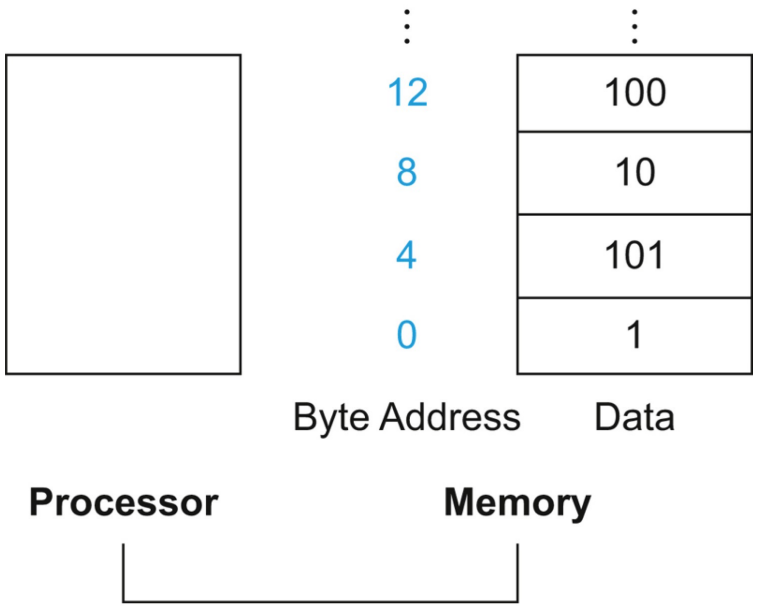
- **Registers are faster to access than memory**
- **Operating on memory data requires loads and stores**
  - More instructions to be executed
- **Compiler must use registers for variables as much as possible**
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Memory Operands

- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **Memory is byte addressed**
  - Just a large, single-dimensional array, with the address acting as the index to that array
  - Words are aligned in memory (address must be a multiple of 4)
- **MIPS is Big Endian**
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address
  - SPIM simulator's byte order is the same as the byte order of the underlying machine that runs the simulator

Accessing memory

- **Arithmetic operations occur only on registers**
- **Data transfer instructions**
  - Transfer data between memory and registers
- **`lw`/`sw` instructions must supply the memory address to access a word in memory**



Addressing Modes

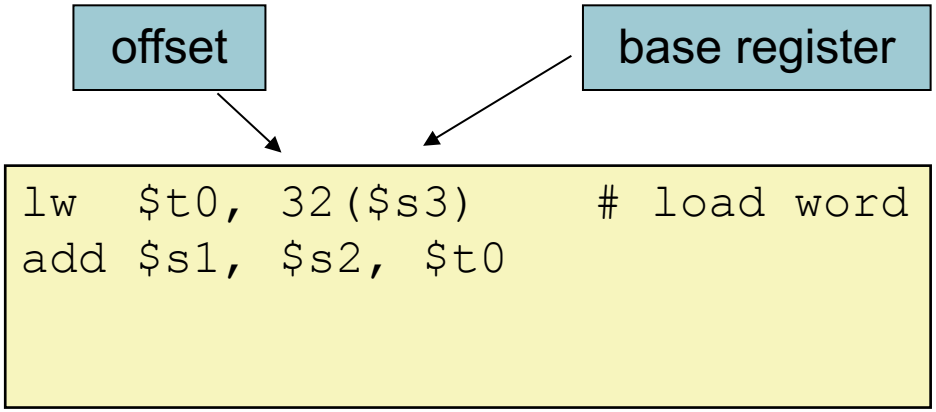
- **Base or displacement addressing**
  - The operand is at the memory location whose address is the sum of a register and a constant in the instruction
- **Immediate addressing**
  - The operand is a constant in the instruction itself (for jumps and conditional branches)

Memory Operand Example 1

- **C code:**

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

- **Compiled MIPS code:**
  - g in `$s1`, h in `$s2`, base address of A in `$s3`
  - Index 8 requires offset of 32
    - 4 bytes per word



Memory Operand Example 2

- **C code:**

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

- **Compiled MIPS code:**
  - h in `$s2`, base address of A in `$s3`
  - Index 8 requires offset of 32

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

Byte/Halfword Operations

`lb rt, offset(rs)` `lh rt, offset(rs)`

- Sign extend to 32 bits in rt

`lbu rt, offset(rs)` `lhu rt, offset(rs)`

- Zero extend to 32 bits in rt

`sb rt, offset(rs)` `sh rt, offset(rs)`

- Store just rightmost byte/halfword

Data Transfer Instructions

`lw $s1, imm($s2)`	$s1 = MEM[$2 + imm]	load word from memory
`lh $s1, imm($s2)`	$s1 = MEM[$2 + imm]	load half word (2 bytes)
`lhu $s1, imm($s2)`	$s1 = MEM[$2 + imm]	load half word unsigned
`lb $s1, imm($s2)`	$s1 = MEM[$2 + imm]	load byte
`lbu $s1, imm($s2)`	$s1 = MEM[$2 + imm]	load byte unsigned
`li $s1, imm`	$s1 = imm	load immediate
`la $s1, L`	$s1 = L	load address
`sw $s1, imm($s2)`	MEM[$2 + imm] = $1	store word to memory
`sh $s1, imm($s2)`	MEM[$2 + imm] = $1	store half word (2 bytes)
`sb $s1, imm($s2)`	MEM[$2 + imm] = $1	store byte
`move $s1, $2`	$s1 = $2	data move

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
- **Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches**
  - Just as it does for calculating data addresses for loads and stores

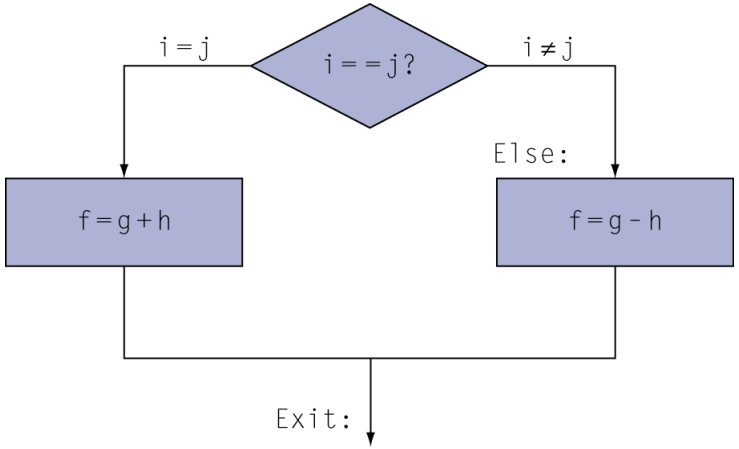
Jump Instructions

- **There is another kind of branch, often called an unconditional branch**
  - These instructions say that the processor always follows the branch
- **The MIPS name for this type of instruction is jump**
  - To distinguish between conditional and unconditional branches,
- `j L1`
  - Unconditional jump to instruction labeled L1

Compiling If Statements

- **C code:**

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



- **Compiled MIPS code:**
  - f, g, … in $s0, $s1, …

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

Assembler calculates addresses

Compiling Loop Statements

- **C code:**

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

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

```
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;
- **Usually used in combination with `beq`, `bne`**

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

Branch Instruction Design

- **Why not `blt`, `bge`, etc in hardware?**
- **Hardware for <, ≥, … slower than =, ≠**
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- **`beq` and `bne` are the common case**
- **Another example of the great idea of making the common case fast**

Assembler Pseudo-instructions

- **Most assembler instructions represent machine instructions one-to-one**

- **Pseudo-instructions**
  - The assembler can also treat common variations of machine instructions as if they were instructions in their own right
  - The hardware does not implement the pseudo-instructions, but their appearance in assembly language simplifies programming

```
move $t0, $t1
li $s1, 20
blt $t0, $t1, L
```

```
add $t0, $zero, $t1
addiu $1, $zero, 20
slt $at, $t0, $t1
bne $at, $zero, L
```

$at (register 1): assembler temporary

Branch Instructions

`beq $1,$2,L`	if($s1 == $2) goto L	branch on equal
`bne $1,$2,L`	if($s1 != $2) goto L	branch on not equal
`blt $1,$2,L`	if($s1 < $2) goto L	branch on less than
`bgt $1,$2,L`	if($s1 > $2) goto L	branch on greater than
`ble $1,$2,L`	if($s1 <= $2) goto L	branch on less than or equal
`bge $1,$2,L`	if($s1 >= $2) goto L	branch on greater than or equal

Jump Instructions

`j`	`imm`	goto imm	jump to target address
`j`	`L`	goto L	jump to target label
`jal`	`L`	$ra=PC+4; goto L	jump and link (for procedure call)
`jr`	`$ra`	goto $ra	jump register (for procedure return)

Concluding Remarks

- **Above the machine level is assembly language, a language that humans can read**
  - Each category of MIPS instructions is associated with constructs that appear in high-level programming languages
- **The assembler translates it into the binary numbers that machines can understand…**
- **…and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware**
  - Hiding details from the higher level is another example of the great idea of abstraction