Introduction to Computer Organization and MIPS Assembly Language

- Part 1 -

Textbook

- Computer Organization & Design:

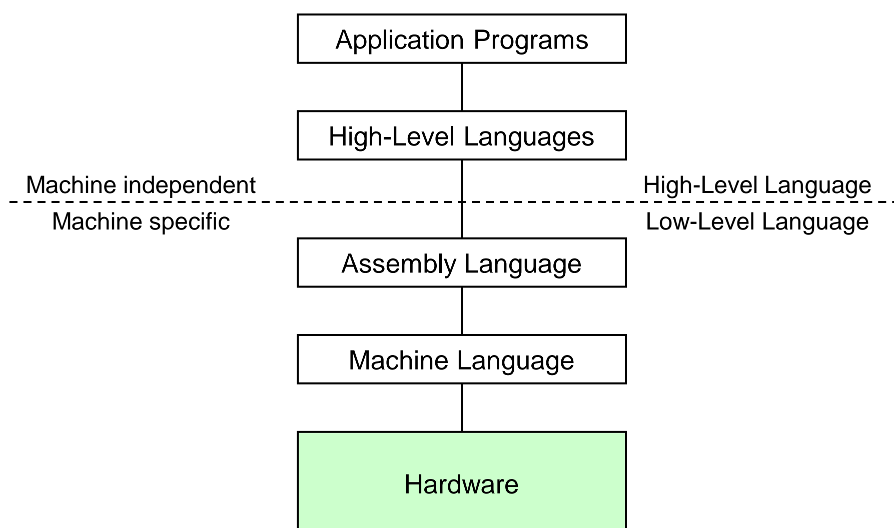
  The Hardware/Software Interface

  - Fifth Edition, 2013
  - David Patterson and John Hennessy
  - Morgan Kaufmann

Some Important Questions to Ask

- What is Assembly Language?
- What is Machine Language?
- How is Assembly related to a high-level language?
- Why Learn Assembly Language?
- What is an Assembler, Linker, and Debugger?

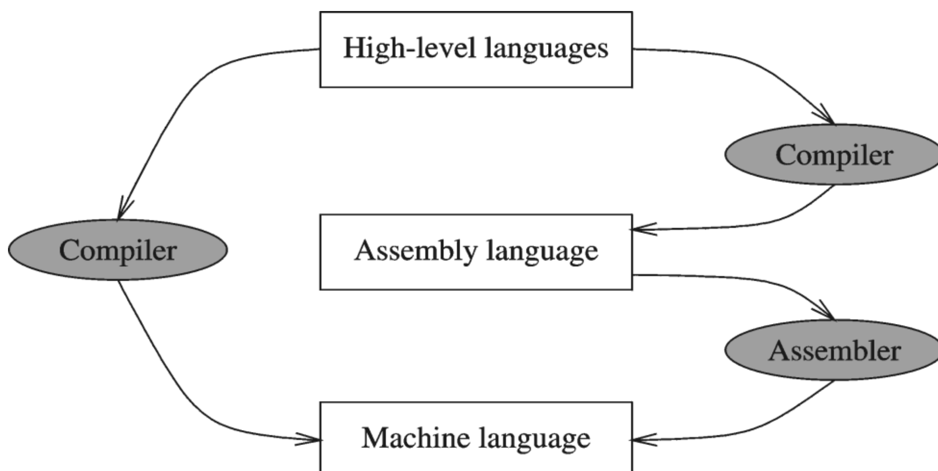
A Hierarchy of Languages



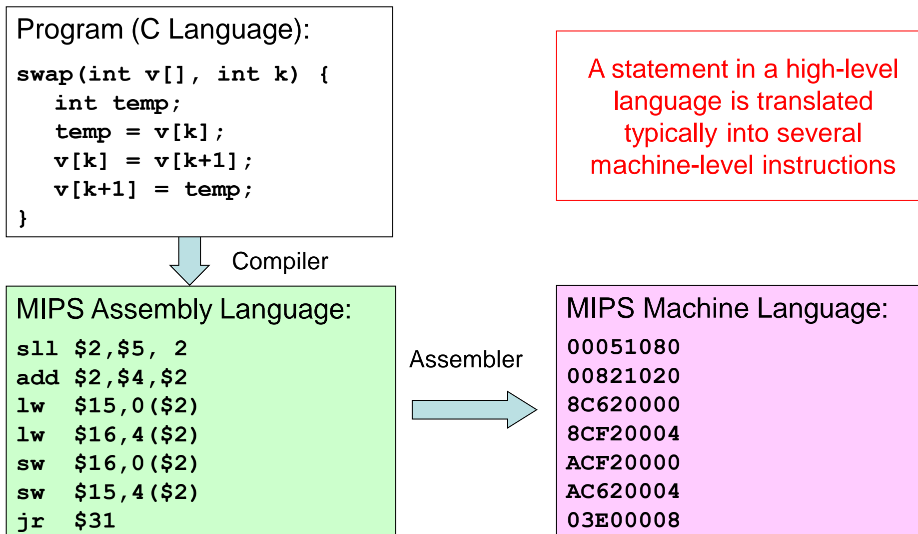
Assembly and Machine Language

- Machine language
  - Native to a processor: executed directly by hardware
  - Instructions consist of binary code: 1s and 0s
- Assembly language
  - Slightly higher-level language
  - Readability of instructions is better than machine language
  - One-to-one correspondence with machine language instructions
- Assemblers translate assembly to machine code
- Compilers translate high-level programs to machine code

Compiler and Assembler



Translating Languages



Advantages of High-Level Languages

❖ Program development is faster
  ✧ High-level statements: fewer instructions to code

❖ Program maintenance is easier
  ✧ For the same above reasons

❖ Programs are portable
  ✧ Contain few machine-dependent details
  ✧ Compiler translates to the target machine language

Why Learn Assembly Language?

- Many reasons:
  - Accessibility to system hardware
  - Space and time efficiency
  - Writing a compiler for a high-level language
- Accessibility to system hardware
  - Assembly Language is useful for implementing system software
  - Also useful for small embedded system applications
- Programming in Assembly Language is harder
  - Requires deep understanding of the processor architecture
  - However, it is very rewarding to system software designers
  - Adds a new perspective on how programs run on real processors

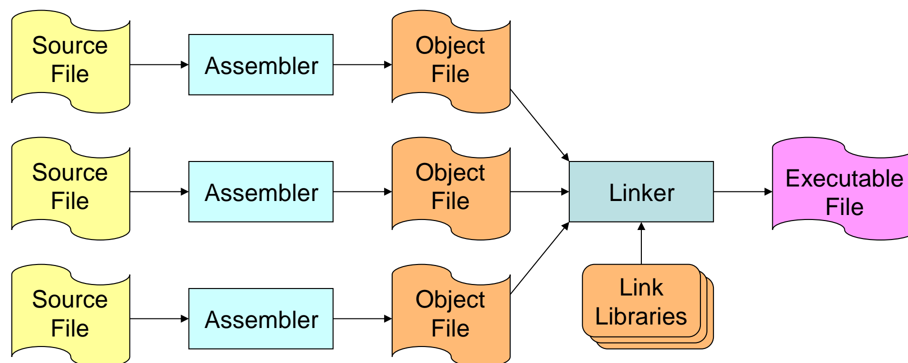
What is Assembly Language?

- Low-level programming language for a computer
- One-to-one correspondence with the machine instructions
- Assembly language is specific to a given processor
- Assembler: converts assembly program into machine code
- Assembly language uses:
  - Mnemonics: to represent the names of low-level machine instructions
  - Labels: to represent the names of variables or memory addresses
  - Directives: to define data and constants
  - Macros: to facilitate the inline expansion of text into other code

Assembly Language Programming Tools

- Editor
  - Allows you to create and edit assembly language source files
- Assembler
  - Converts assembly language programs into object files
  - Object files contain the machine instructions
- Linker
  - Combines object files created by the assembler with link libraries
  - Produces a single executable program
- Debugger
  - Allows you to trace the execution of a program
  - Allows you to view machine instructions, memory, and registers

Assemble and Link Process



- A program may consist of multiple source files
- Assembler translates each source file into an object file
- Linker links all object files together and with link libraries
- The result executable file can run directly on the processor

Classes of Computers

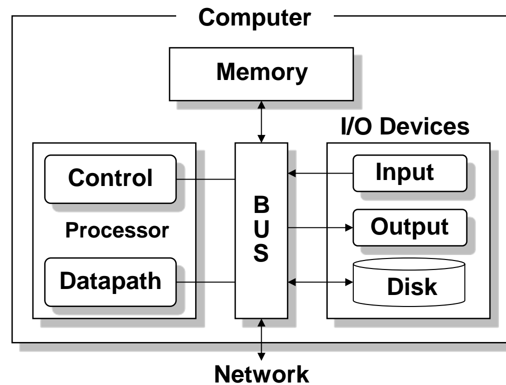
- Personal computers
  - General purpose, variety of software, subject to cost/performance
- Server computers
  - Network based, high capacity, performance, and reliability
  - Range from small servers to building sized
- Supercomputers
  - High-end scientific and engineering calculations
  - Highest capability but only a small fraction of the computer market
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

Classes of Computers (cont'd)

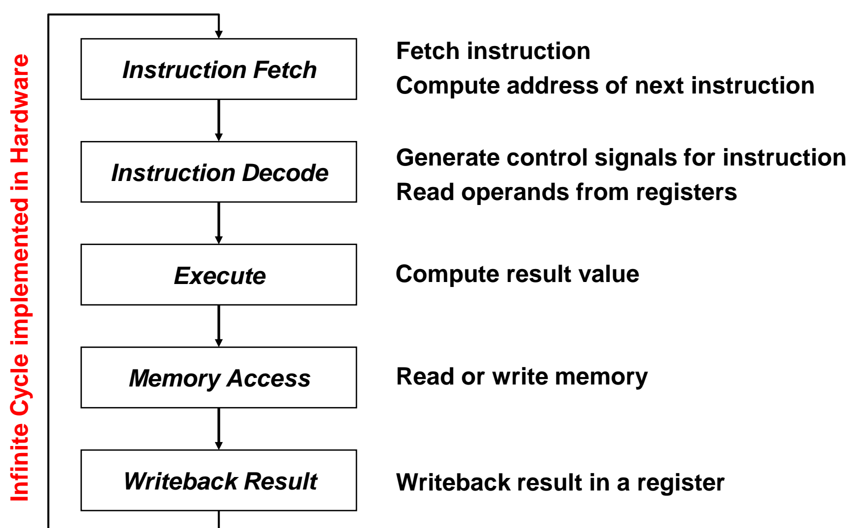
- Personal Mobile Device (PMD)
  - Battery operated
  - Connects to the Internet
  - Low price: hundreds of dollars
  - Smart phones, tablets, electronic glasses
- Cloud Computing
  - Warehouse Scale Computers (WSC)
  - Software, Platform, and Infrastructure as a Service
  - However, security concerns of storing "sensitive data" in "the cloud"
  - Examples: Amazon and Google

Components of a Computer System

- Processor
  - Datapath and Control
- Memory & Storage
  - Main Memory
  - Disk Storage
- Input / Output devices
  - User-interface devices
  - Network adapters
    - For communicating with other computers
- Bus: Interconnects processor to memory and I/O
- Essentially the same components for all kinds of computers



Fetch - Execute Cycle



Instruction Set Architecture (ISA)

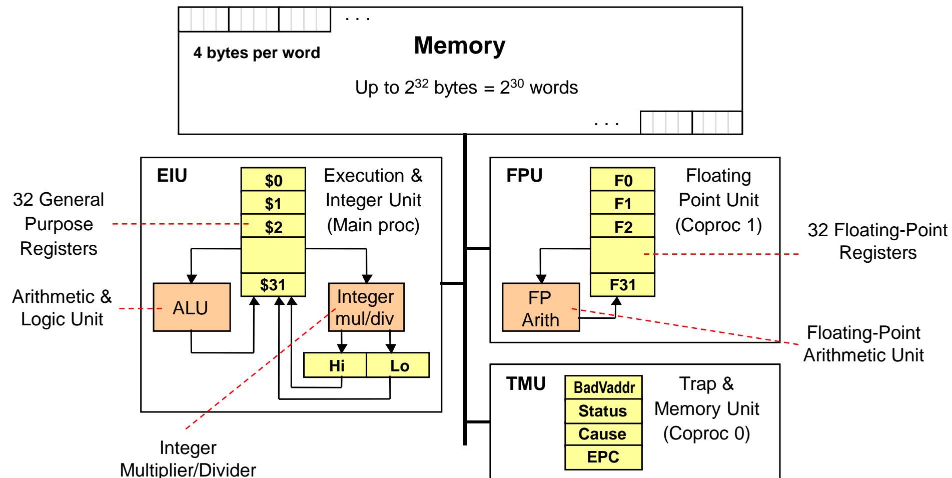
- Critical Interface between software and hardware
- An ISA includes the following …
  - Instructions and Instruction Formats
  - Data Types, Encodings, and Representations
  - Programmable Storage: Registers and Memory
  - Addressing Modes: to address Instructions and Data
  - Handling Exceptional Conditions (like overflow)

Examples	(Versions)	Introduced in
Intel	(8086, 80386, Pentium, Core, ...)	1978
MIPS	(MIPS I, II, …, MIPS32, MIPS64)	1986
ARM	(version 1, 2, …)	1985

Instructions

- Instructions are the language of the machine
- We will study the MIPS instruction set architecture
  - Known as **Reduced Instruction Set Computer (RISC)**
  - Elegant and relatively simple design
  - Similar to RISC architectures developed in mid-1980's and 90's
  - Popular, used in many products
    - Silicon Graphics, ATI, Cisco, Sony, etc.
- Alternative to: Intel x86 architecture
  - Known as **Complex Instruction Set Computer (CISC)**

Overview of the MIPS Architecture



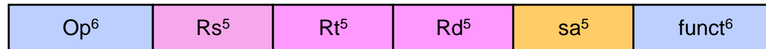
MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - All registers are 32-bit wide in the MIPS 32-bit architecture
  - Software defines names for registers to standardize their use
  - Assembler can refer to registers by name or by number ($ notation)

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

Instruction Formats

- All instructions are 32-bit wide, Three instruction formats:
- Register (R-Type)
  - Register-to-register instructions
  - Op: operation code specifies the format of the instruction



- Immediate (I-Type)
  - 16-bit immediate constant is part in the instruction

- Jump (J-Type)
  - Used by jump instructions

Assembly Language Instructions

- Assembly language instructions have the format:

```
[label:]   mnemonic   [operands]   [#comment]
```

- Label: (optional)
  - Marks the address of a memory location, must have a colon
  - Typically appear in data and text segments
- Mnemonic
  - Identifies the operation (e.g. `add`, `sub`, etc.)
- Operands
  - Specify the data required by the operation
  - Operands can be registers, memory variables, or constants
  - Most instructions have three operands

```
L1:   addiu $t0, $t0, 1       #increment $t0
```

Assembly Language Statements

- ❖ Three types of statements in assembly language
  - ✧ Typically, one statement should appear on a line

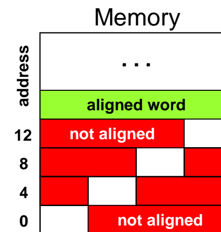
1. Executable Instructions
   - ✧ Generate machine code for the processor to execute at runtime
   - ✧ Instructions tell the processor what to do
2. Pseudo-Instructions and Macros
   - ✧ Translated by the assembler into real instructions
   - ✧ Simplify the programmer task
3. Assembler Directives
   - ✧ Provide information to the assembler while translating a program
   - ✧ Used to define segments, allocate memory variables, etc.
   - ✧ Non-executable: directives are not part of the instruction set

Comments

- ❖ Single-line comment
  - ✧ Begins with a hash symbol **#** and terminates at end of line
- ❖ Comments are very important!
  - ✧ Explain the program's purpose
  - ✧ When it was written, revised, and by whom
  - ✧ Explain data used in the program, input, and output
  - ✧ Explain instruction sequences and algorithms used
  - ✧ Comments are also required at the beginning of every procedure
    - ▪ Indicate input parameters and results of a procedure
    - ▪ Describe what the procedure does

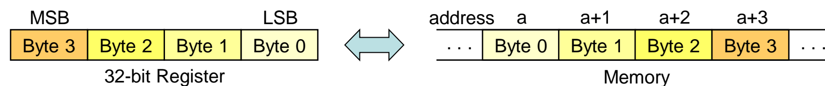
Memory Alignment

- Memory is viewed as an **addressable array of bytes**
- **Byte Addressing**: address points to a byte in memory
- However, words occupy 4 consecutive bytes in memory
  - MIPS instructions and integers occupy 4 bytes
- **Memory Alignment**:
  - Address must be multiple of size
  - Word address should be a multiple of **4**
  - Double-word address should be a multiple of **8**

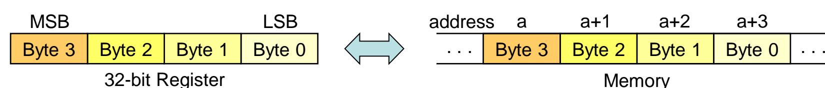


Byte Ordering (Endianness)

- Processors can order bytes within a word in two ways
- Little Endian Byte Ordering
  - Memory address = Address of **least significant byte**
  - Example: Intel IA-32



- Big Endian Byte Ordering
  - Memory address = Address of **most significant byte**
  - Example: SPARC architecture



- MIPS can operate with both byte orderings

Instruction Categories

- Integer Arithmetic
  - Arithmetic, logic, and shift instructions
- Data Transfer
  - Load and store instructions that access memory
  - Data movement and conversions
- Jump and Branch
  - Flow-control instructions that alter the sequential sequence

R-Type Instruction Format

- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction
- **funct**: function code – extends the opcode
  - Up to 2^6 = 64 functions can be defined for the same opcode
  - MIPS uses opcode 0 to define many R-type instructions
- Three Register Operands (common to many instructions)
  - **Rs**, **Rt**: first and second source operands
  - **Rd**: destination operand
  - **sa**: the shift amount used by shift instructions

R-Type Integer Add and Subtract

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
add $t1, $t2, $t3	$t1 = $t2 + $t3	0	$t2	$t3	$t1	0	0x20
addu $t1, $t2, $t3	$t1 = $t2 + $t3	0	$t2	$t3	$t1	0	0x21
sub $t1, $t2, $t3	$t1 = $t2 - $t3	0	$t2	$t3	$t1	0	0x22
subu $t1, $t2, $t3	$t1 = $t2 - $t3	0	$t2	$t3	$t1	0	0x23

- **add, sub: arithmetic overflow causes an exception**
  - In case of overflow, result is not written to destination register
- **addu, subu: arithmetic overflow is ignored**
- **addu, subu**: compute the same result as **add, sub**
- Many programming languages ignore overflow
  - The **+** operator is translated into **addu**
  - The **–** operator is translated into **subu**

Using Add / Subtract Instructions

- Consider the translation of: `f = (g+h)-(i+j)`
- Programmer / Compiler allocates registers to variables
- Given that: `$t0=f, $t1=g, $t2=h, $t3=i, and $t4=j`
- Called temporary registers: `$t0=$8, $t1=$9, …`
- Translation of: `f = (g+h)-(i+j)`

```
addu $t5, $t1, $t2   # $t5 = g + h
addu $t6, $t3, $t4   # $t6 = i + j
subu $t0, $t5, $t6   # f = (g+h)-(i+j)
```

- Assembler translates `addu $t5,$t1,$t2` into binary code

Op	$t1	$t2	$t5	sa	addu
000000	01001	01010	01101	00000	100001

Logic Bitwise Operations

❖ Logic bitwise operations: **and, or, xor, nor**

x	*y*	*x* and *y*
0	0	0
0	1	0
1	0	0
1	1	1

x	*y*	*x* or *y*
0	0	0
0	1	1
1	0	1
1	1	1

x	*y*	*x* xor *y*
0	0	0
0	1	1
1	0	1
1	1	0

x	*y*	*x* nor *y*
0	0	1
0	1	0
1	0	0
1	1	0

❖ AND instruction is used to clear bits: `x and 0 → 0`

❖ OR instruction is used to set bits: `x or 1 → 1`

❖ XOR instruction is used to toggle bits: `x xor 1 → not x`

❖ NOT instruction is not needed, why?

`not $t1, $t2` is equivalent to: `nor $t1, $t2, $t2`

Logic Bitwise Instructions

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
and $t1, $t2, $t3	$t1 = $t2 & $t3	0	$t2	$t3	$t1	0	0x24
or $t1, $t2, $t3	$t1 = $t2 | $t3	0	$t2	$t3	$t1	0	0x25
xor $t1, $t2, $t3	$t1 = $t2 ^ $t3	0	$t2	$t3	$t1	0	0x26
nor $t1, $t2, $t3	$t1 = ~($t2|$t3)	0	$t2	$t3	$t1	0	0x27

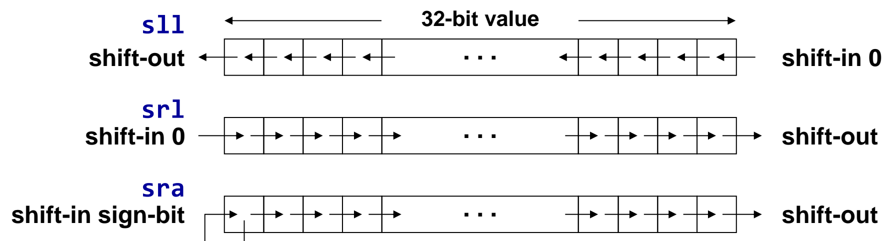
❖ Examples:

Given: `$t1 = 0xabcd1234` and `$t2 = 0xffff0000`

```
and $t0, $t1, $t2        # $t0 = 0xabcd0000
or  $t0, $t1, $t2        # $t0 = 0xffff1234
xor $t0, $t1, $t2        # $t0 = 0x54321234
nor $t0, $t1, $t2        # $t0 = 0x0000edcb
```

Shift Operations

- Shifting is to move the 32 bits of a number left or right
- `sll` means **shift left logical** (insert zero from the right)
- `srl` means **shift right logical** (insert zero from the left)
- `sra` means **shift right arithmetic** (insert sign-bit)
- The **5-bit shift amount** field is used by these instructions



Shift Instructions

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
sll $t1,$t2,10	$t1 = $t2 << 10	0	0	$t2	$t1	10	0
srl $t1,$t2,10	$t1 = $t2 >>> 10	0	0	$t2	$t1	10	2
sra $t1,$t2,10	$t1 = $t2 >> 10	0	0	$t2	$t1	10	3
sllv $t1,$t2,$t3	$t1 = $t2 << $t3	0	$t3	$t2	$t1	0	4
srlv $t1,$t2,$t3	$t1 = $t2 >>>$t3	0	$t3	$t2	$t1	0	6
srav $t1,$t2,$t3	$t1 = $t2 >> $t3	0	$t3	$t2	$t1	0	7

- `sll, srl, sra:` **shift by a constant amount**
  - The shift amount (**sa**) field specifies a number between 0 and 31
- `sllv, srlv, srav:` **shift by a variable amount**
  - A source register specifies the variable shift amount between 0 and 31
  - Only the lower 5 bits of the source register is used as the shift amount

Shift Instruction Examples

❖ Given that: **$t2 = 0xabcd1234 and $t3 = 16**

```
sll  $t1, $t2, 8          $t1 = 0xcd123400
srl  $t1, $t2, 4          $t1 = 0x0abcd123
sra  $t1, $t2, 4          $t1 = 0xfabcd123
srlv $t1, $t2, $t3        $t1 = 0x0000abcd
```

Op	Rs = $t3	Rt = $t2	Rd = $t1	sa	srlv
000000	01011	01010	01001	00000	000110

Binary Multiplication

❖ Shift Left Instruction (`sll`) can perform multiplication

✧ When the multiplier is a power of 2

❖ You can factor any binary number into powers of 2

❖ Example: multiply **$t0** by **36**

$t0*36 = $t0*(4 + 32) = $t0*4 + $t0*32

```
sll  $t1, $t0, 2        # $t1 = $t0 * 4
sll  $t2, $t0, 5        # $t2 = $t0 * 32
addu $t3, $t1, $t2      # $t3 = $t0 * 36
```

Your Turn . . .

Multiply **`$t0`** by **26**, using shift and add instructions

Hint: **26 = 2 + 8 + 16**

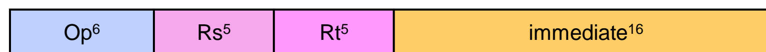
```
sll   $t1, $t0, 1         # $t1 = $t0 * 2
sll   $t2, $t0, 3         # $t2 = $t0 * 8
sll   $t3, $t0, 4         # $t3 = $t0 * 16
addu  $t4, $t1, $t2       # $t4 = $t0 * 10
addu  $t5, $t4, $t3       # $t5 = $t0 * 26
```

Multiply **`$t0`** by **31**, Hint: **31 = 32 – 1**

```
sll   $t1, $t0, 5         # $t1 = $t0 * 32
subu  $t2, $t1, $t0       # $t2 = $t0 * 31
```

I-Type Instruction Format

- Constants are used quite frequently in programs
  - The R-type shift instructions have a 5-bit shift amount constant
  - What about other instructions that need a constant?
- I-Type: Instructions with Immediate Operands



- 16-bit immediate constant is stored inside the instruction
  - Rs is the source register number
  - Rt is now the destination register number (for R-type it was Rd)
- Examples of I-Type ALU Instructions:
  - Add immediate: `addi $t1, $t2, 5   # $t1 = $t2 + 5`
  - OR immediate: `ori  $t1, $t2, 5   # $t1 = $t2 | 5`

I-Type ALU Instructions

Instruction	Meaning	Op	Rs	Rt	Immediate
`addi $t1, $t2, 25`	`$t1 = $t2 + 25`	0x8	$t2	$t1	25
`addiu $t1, $t2, 25`	`$t1 = $t2 + 25`	0x9	$t2	$t1	25
`andi $t1, $t2, 25`	`$t1 = $t2 & 25`	0xc	$t2	$t1	25
`ori $t1, $t2, 25`	`$t1 = $t2 | 25`	0xd	$t2	$t1	25
`xori $t1, $t2, 25`	`$t1 = $t2 ^ 25`	0xe	$t2	$t1	25
`lui $t1, 25`	`$t1 = 25 << 16`	0xf	0	$t1	25

- **addi**: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- **addiu**: same operation as **addi** but overflow is ignored
- Immediate constant for **addi** and **addiu** is signed
  - No need for **subi** or **subiu** instructions
- Immediate constant for **andi**, **ori**, **xori** is unsigned

Examples of I-Type ALU Instructions

- Given that registers **$t0, $t1, $t2** are used for **A, B, C**

Expression	Equivalent MIPS Instruction
`A = B + 5;`	`addiu $t0, $t1, 5`
`C = B – 1;`	`addiu $t2, $t1, -1`
`A = B & 0xf;`	`andi $t0, $t1, 0xf`
`C = B | 0xf;`	`ori $t2, $t1, 0xf`
`C = 5;`	`addiu $t2, $zero, 5`
`A = B;`	`addiu $t0, $t1, 0`

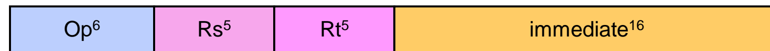
Op = addiu	Rs = $t1	Rt = $t2	-1 = 0b1111111111111111

No need for **subiu**, because **addiu** has signed immediate

Register **$zero** has always the value **0**

32-bit Constants

- I-Type instructions can have only 16-bit constants



- What if we want to load a 32-bit constant into a register?
- **Can't have a 32-bit constant in I-Type instructions ☹**
  - The sizes of all instructions are fixed to 32 bits
- **Solution: use two instructions instead of one ☺**
- Suppose we want: `$t1 = 0xAC5165D9` (32-bit constant)

  `lui`: **load upper immediate**

		Upper 16 bits	Lower 16 bits
`lui $t1, 0xAC51`	`$t1`	`0xAC51`	`0x0000`
`ori $t1, $t1, 0x65D9`	`$t1`	`0xAC51`	`0x65D9`

Pseudo-Instructions

- Introduced by the assembler as if they were real instructions
- Facilitate assembly language programming

Pseudo-Instruction	Equivalent MIPS Instruction
`move $t1, $t2`	`addu $t1, $t2, $zero`
`not $t1, $t2`	`nor $t1, $t2, $zero`
`neg $t1, $t2`	`sub $t1, $zero, $t2`
`li $t1, -5`	`addiu $t1, $zero, -5`
`li $t1, 0xabcd1234`	`lui $t1, 0xabcd` `ori $t1, $t1, 0x1234`

The MARS tool has a long list of pseudo-instructions

Control Flow

- High-level programming languages provide constructs:
  - To make decisions in a program: IF-ELSE
  - To repeat the execution of a sequence of instructions: LOOP
- The ability to make decisions and repeat a sequence of instructions distinguishes a computer from a calculator
- All computer architectures provide control flow instructions
- Essential for making decisions and repetitions
- These are the **conditional branch** and **jump** instructions

MIPS Conditional Branch Instructions

- MIPS **compare and branch** instructions:

```
beq Rs, Rt, label      if (Rs == Rt) branch to label
bne Rs, Rt, label      if (Rs != Rt) branch to label
```

- MIPS **compare to zero & branch** instructions:

  Compare to zero is used frequently and implemented efficiently

```
bltz Rs, label      if (Rs < 0) branch to label
bgtz Rs, label      if (Rs > 0) branch to label
blez Rs, label      if (Rs <= 0) branch to label
bgez Rs, label      if (Rs >= 0) branch to label
```

- `beqz` and `bnez` are defined as pseudo-instructions.

Branch Instruction Format

❖ Branch Instructions are of the I-type Format:

Instruction	I-Type Format			
`beq Rs, Rt, label`	Op = 4	Rs	Rt	16-bit Offset
`bne Rs, Rt, label`	Op = 5	Rs	Rt	16-bit Offset
`blez Rs, label`	Op = 6	Rs	0	16-bit Offset
`bgtz Rs, label`	Op = 7	Rs	0	16-bit Offset
`bltz Rs, label`	Op = 1	Rs	0	16-bit Offset
`bgez Rs, label`	Op = 1	Rs	1	16-bit Offset

❖ The branch instructions modify the **PC register** only

❖ **PC-Relative addressing**:

If (branch is taken) **PC = PC + 4 + 4×offset** else **PC = PC+4**

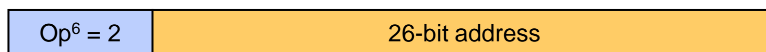
Unconditional Jump Instruction

❖ The unconditional Jump instruction has the following syntax:

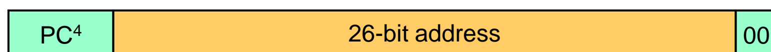
```
j    label     # jump to label
  . . .
label:
```

❖ The jump instruction is **always taken**

❖ The Jump instruction is of the J-type format:



❖ The jump instruction modifies the program counter PC:



❖ The upper 4 bits of the PC are unchanged

multiple of 4

Translating an IF Statement

- Consider the following IF statement:

```
if (a == b) c = d + e; else c = d – e;
```

Given that **a, b, c, d, e** are in **$t0 … $t4** respectively

- How to translate the above IF statement?

```
        bne     $t0, $t1, else
        addu    $t2, $t3, $t4
        j       next
else:   subu    $t2, $t3, $t4
next:   . . .
```

Logical AND Expression

- Programming languages use **short-circuit evaluation**
- If first condition is **false**, second condition is **skipped**

```
if (($t1 > 0) && ($t2 < 0)) {$t3++;}
```

```
# One Possible Translation ...
    bgtz  $t1, L1        # first condition
    j     next           # skip if false
L1: bltz  $t2, L2        # second condition
    j     next           # skip if false
L2: addiu $t3, $t3, 1    # both are true
next:
```

Better Translation of Logical AND

```
if (($t1 > 0) && ($t2 < 0)) {$t3++;}
```

Allow the program to **fall through** to second condition

!($t1 > 0) is equivalent to **($t1 <= 0)**

!($t2 < 0) is equivalent to **($t2 >= 0)**

Number of instructions is reduced from **5** to **3**

```
# Better Translation ...
   blez   $t1, next        # 1st condition false?
   bgez   $t2, next        # 2nd condition false?
   addiu  $t3, $t3, 1      # both are true
next:
```

Logical OR Expression

- **Short-circuit evaluation** for logical OR
- If first condition is **true**, second condition is **skipped**

```
if (($t1 > 0) || ($t2 < 0)) {$t3++;}
```

- Use **fall-through** to keep the code as short as possible

```
    bgtz  $t1, L1        # 1st condition true?
    bgez  $t2, next      # 2nd condition false?
L1: addiu $t3, $t3, 1    # increment $t3
next:
```

Compare Instructions

❖ MIPS also provides **set less than** instructions

`slt   Rd, Rs, Rt`	if (Rs < Rt) Rd = 1 else Rd = 0
`sltu  Rd, Rs, Rt`	**unsigned <**
`slti  Rt, Rs, imm`	if (Rs < imm) Rt = 1 else Rt = 0
`sltiu Rt, Rs, imm`	**unsigned <**

❖ **Signed / Unsigned** comparisons compute different results

Given that: `$t0 = 1` and `$t1 = -1 = 0xffffffff`

`slt   $t2, $t0, $t1`   computes   `$t2 = 0`

`sltu  $t2, $t0, $t1`   computes   `$t2 = 1`

Compare Instruction Formats

Instruction	Meaning	Format					
`slt   Rd, Rs, Rt`	Rd=(Rs $<_s$ Rt)?1:0	Op=0	Rs	Rt	Rd	0	0x2a
`sltu  Rd, Rs, Rt`	Rd=(Rs $<_u$ Rt)?1:0	Op=0	Rs	Rt	Rd	0	0x2b
`slti  Rt, Rs, im`	Rt=(Rs $<_s$ im)?1:0	0xa	Rs	Rt	16-bit immediate		
`sltiu Rt, Rs, im`	Rt=(Rs $<_u$ im)?1:0	0xb	Rs	Rt	16-bit immediate		

❖ The other comparisons are defined as pseudo-instructions:

`seq, sne, sgt, sgtu, sle, sleu, sge, sgeu`

Pseudo-Instruction	Equivalent MIPS Instructions
`sgt  $t2, $t0, $t1`	`slt   $t2, $t1, $t0`
`seq  $t2, $t0, $t1`	`subu  $t2, $t0, $t1` `sltiu $t2, $t2, 1`

Pseudo-Branch Instructions

❖ MIPS hardware does NOT provide the following instructions:

blt, bltu	branch if less than	(signed / unsigned)
ble, bleu	branch if less or equal	(signed / unsigned)
bgt, bgtu	branch if greater than	(signed / unsigned)
bge, bgeu	branch if greater or equal	(signed / unsigned)

❖ MIPS assembler defines them as pseudo-instructions:

Pseudo-Instruction	Equivalent MIPS Instructions
`blt $t0, $t1, label`	`slt $at, $t0, $t1` `bne $at, $zero, label`
`ble $t0, $t1, label`	`slt $at, $t1, $t0` `beq $at, $zero, label`

$at (**$1**) is the **assembler temporary register**

Using Pseudo-Branch Instructions

❖ Translate the IF statement to assembly language

❖ **$t1** and **$t2** values are **unsigned**

```
if($t1 <= $t2) {
  $t3 = $t4;
}
```

```
   bgtu  $t1, $t2, L1
   move  $t3, $t4
L1:
```

❖ **$t3**, **$t4**, and **$t5** values are **signed**

```
if (($t3 <= $t4) &&
    ($t4 >= $t5)) {
  $t3 = $t4 + $t5;
}
```

```
   bgt   $t3, $t4, L1
   blt   $t4, $t5, L1
   addu  $t3, $t4, $t5
L1:
```

Conditional Move Instructions

Instruction	Meaning	R-Type Format					
movz Rd, Rs, Rt	if (Rt==0) Rd=Rs	Op=0	Rs	Rt	Rd	0	0xa
movn Rd, Rs, Rt	if (Rt!=0) Rd=Rs	Op=0	Rs	Rt	Rd	0	0xb

```
if ($t0 == 0) {$t1=$t2+$t3;} else {$t1=$t2-$t3;}
```

```
     bne   $t0, $0, L1
     addu  $t1, $t2, $t3
     j     L2
L1:  subu  $t1, $t2, $t3
L2:  . . .
```

```
addu  $t1, $t2, $t3
subu  $t4, $t2, $t3
movn  $t1, $t4, $t0
. . .
```

- ❖ Conditional move can eliminate branch & jump instructions

Arrays

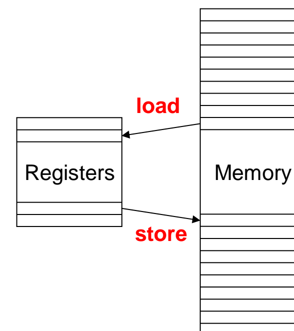
- ❖ In a high-level programming language, an array is a homogeneous data structure with the following properties:
  - ✧ All array elements are of the same type and size
  - ✧ Once an array is allocated, its size cannot be modified
  - ✧ The base address is the address of the first array element
  - ✧ The array elements can be indexed
  - ✧ The address of any array element can be computed
- ❖ In assembly language, an array is just a block of memory
- ❖ In fact, all objects are simply blocks of memory
- ❖ The memory block can be allocated statically or dynamically

Load and Store Instructions

❖ Instructions that transfer data between memory & registers

❖ Programs include variables such as arrays and objects

❖ These variables are stored in memory



❖ Load Instruction:

✧ Transfers data from memory to a register

❖ Store Instruction:

✧ Transfers data from a register to memory

❖ Memory address must be specified by load and store

Load and Store Word

❖ Load Word Instruction (Word = 4 bytes in MIPS)

```
lw Rt, imm(Rs)     # Rt ← MEMORY[Rs+imm]
```

❖ Store Word Instruction

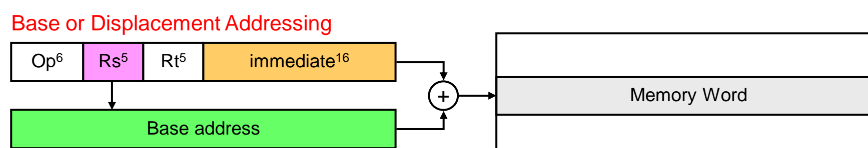
```
sw Rt, imm(Rs)     # Rt → MEMORY[Rs+imm]
```

❖ **Base / Displacement addressing** is used

✧ Memory Address = Rs (**base**) + Immediate (**displacement**)

✧ Immediate^{16} is sign-extended to have a signed displacement

Base or Displacement Addressing



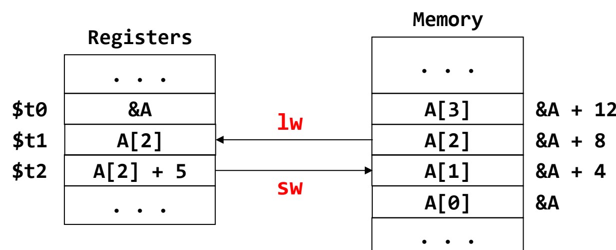
Example on Load & Store

❖ Translate: `A[1] = A[2] + 5` (`A` is an array of words)

❖ Given that the address of array `A` is stored in register `$t0`

```
lw      $t1, 8($t0)      # $t1 = A[2]
addiu   $t2, $t1, 5      # $t2 = A[2] + 5
sw      $t2, 4($t0)      # A[1] = $t2
```

❖ Index of `A[2]` and `A[1]` should be multiplied by 4. Why?



Addressing Modes

❖ Where are the operands?

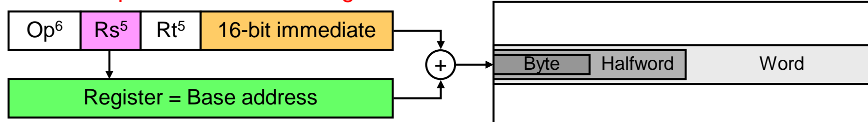
❖ How memory addresses are computed?

Immediate Addressing

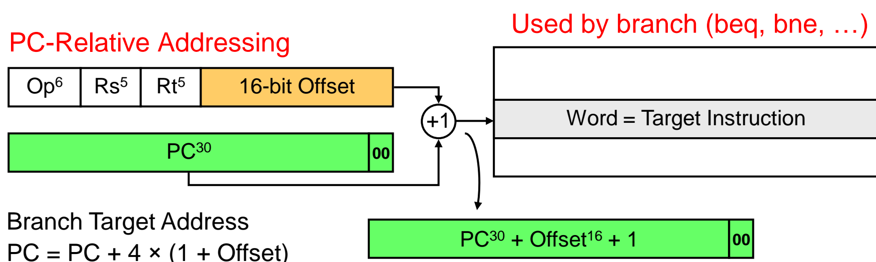
Register Addressing

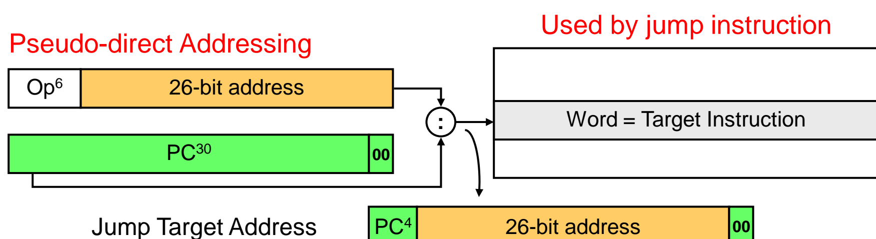
Base / Displacement Addressing

Memory Addressing (load/store)



Branch / Jump Addressing Modes





Integer Multiplication in MIPS

❖ Multiply instructions

✧ `mult Rs, Rt` **Signed multiplication**

✧ `multu Rs, Rt` **Unsigned multiplication**

❖ 32-bit multiplication produces a 64-bit Product

❖ Separate pair of 32-bit registers

✧ **HI = high-order 32-bit of product**

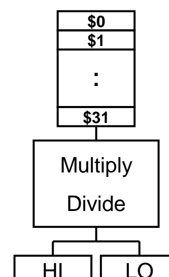
✧ **LO = low-order 32-bit of product**

❖ MIPS also has a special `mul` instruction

✧ `mul Rd, Rs, Rt` `Rd = Rs × Rt`

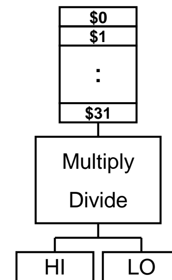
✧ Copy **LO** into destination register **Rd**

✧ Useful when the product is small (32 bits) and **HI** is not needed



Integer Division in MIPS

- Divide instructions
  - `div Rs, Rt` **Signed division**
  - `divu Rs, Rt` **Unsigned division**
- Division produces quotient and remainder
- Separate pair of 32-bit registers
  - **HI = 32-bit remainder**
  - **LO = 32-bit quotient**
  - If divisor is 0 then result is **unpredictable**
- Moving data from **HI, LO** to MIPS registers
  - `mfhi Rd` (**Rd = HI**)
  - `mflo Rd` (**Rd = LO**)



Integer Multiply and Divide Instructions

Instruction	Meaning	Format					
mult Rs, Rt	HI, LO = Rs $\times_s$ Rt	Op = 0	Rs	Rt	0	0	0x18
multu Rs, Rt	HI, LO = Rs $\times_u$ Rt	Op = 0	Rs	Rt	0	0	0x19
mul Rd, Rs, Rt	Rd = Rs $\times_s$ Rt	0x1c	Rs	Rt	Rd	0	2
div Rs, Rt	HI, LO = Rs $/_s$ Rt	Op = 0	Rs	Rt	0	0	0x1a
divu Rs, Rt	HI, LO = Rs $/_u$ Rt	Op = 0	Rs	Rt	0	0	0x1b
mfhi Rd	Rd = HI	Op = 0	0	0	Rd	0	0x10
mflo Rd	Rd = LO	Op = 0	0	0	Rd	0	0x12
mthi Rs	HI = Rs	Op = 0	Rs	0	0	0	0x11
mtlo Rs	LO = Rs	Op = 0	Rs	0	0	0	0x13

$\times_s$ = Signed multiplication, $\times_u$ = Unsigned multiplication

$/_s$ = Signed division, $/_u$ = Unsigned division

Summary of RISC Design

- All instructions are of the same size
- Few instruction formats
- All arithmetic and logic operations are register to register
  - Operands are read from registers
  - Result is stored in a register
- General purpose registers for data and memory addresses
- Memory access only via load and store instructions
  - Load and store: bytes, half words, and words
- Few simple addressing modes