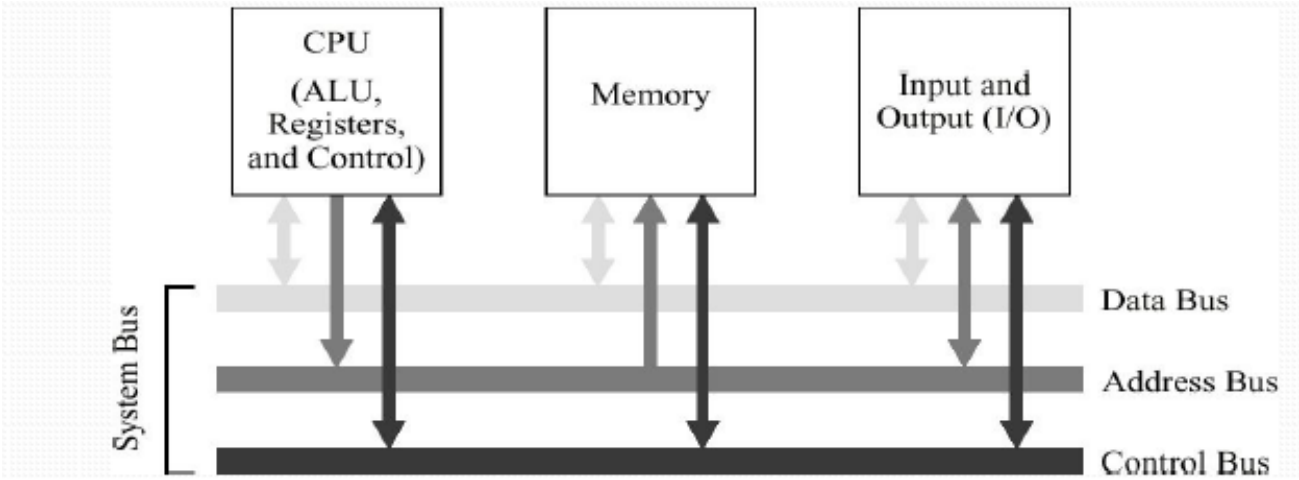
Bus Interconnection الربط بين الحافلات

- Communication among components is handled by a shared pathway called the **system bus**, which is made up of the data bus, the address bus, and the control bus. There is also a power bus, and some architecture may also have a separate I/O bus.
- يتم التعامل مع الاتصالات بين المكونات من خلال مسار مشترك يسمى حافلة النظام، والتي تتكون من حافلة البيانات وحافلة العناوين وحافلة التحكم. هناك أيضًا حافلة طاقة، وقد تحتوي بعض البنى أيضًا على حافلة إدخال/إخراج منفصلة.
- Multiple devices are connected to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus.
- يتم توصيل أجهزة متعددة بالحافلة، وتكون الإشارة المرسلة بواسطة أي جهاز متاحة للاستقبال بواسطة جميع الأجهزة الأخرى المتصلة بالحافلة.
- If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.
- إذا قام جهازان بالإرسال خلال نفس الفترة الزمنية، فسوف تتداخل إشاراتهما وتصبح مشوهة. وبالتالي، لا يمكن إلا لجهاز واحد في كل مرة الإرسال بنجاح.



Bus Structure: Data Bus بنية الناقل: ناقل البيانات

- **Data Bus:** provide a path for moving data among system modules. The data bus may consist of 32, 64, 128, or more separate lines,
- ناقل البيانات: يوفر مسارًا لنقل البيانات بين وحدات النظام. قد يتكون ناقل البيانات من ٣٢ أو ٦٤ أو ١٢٨ خطًا منفصلًا أو أكثر،
  - Number of lines being referred to as the *width* of the data bus. Because each line can carry only 1 bit at a time. The width of the data bus is a key factor in determining overall system performance
  - ويشار إلى عدد الخطوط على أنه عرض ناقل البيانات. لأن كل خط يمكنه حمل بت واحد فقط في المرة الواحدة. يعد عرض ناقل البيانات عاملاً رئيسيًا في تحديد الأداء العام للنظام
  - Number of lines determines how many bits can be transferred at a time. For example, if the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle.
  - يحدد عدد الخطوط عدد البتات التي يمكن نقلها في المرة الواحدة. على سبيل المثال، إذا كان عرض ناقل البيانات ٣٢ بتًا وكان طول كل تعليمة ٦٤ بتًا، فيجب على المعالج الوصول إلى وحدة الذاكرة مرتين خلال كل دورة تعليمة.

Bus Structure: Address Bus
هيكل الحافلة: عنوان الحافلة

The **address lines** are used to designate the source or destination of the data on the data bus.

تُستخدم خطوط العناوين لتحديد مصدر أو وجهة البيانات على ناقل البيانات.

- If the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines.
- إذا رغب المعالج في قراءة كلمة (٨ أو ١٦ أو ٣٢ بت) من البيانات من الذاكرة، فإنه يضع عنوان الكلمة المطلوبة على خطوط العناوين.

- Clearly, the width of the **address bus** determines the maximum possible memory capacity of the system. Address bus of width=n, can address up to 2^n memory location.
- من الواضح أن عرض ناقل العناوين يحدد أقصى سعة ذاكرة ممكنة للنظام. ناقل العناوين بعرض = n، يمكنه معالجة ما يصل إلى ٢ nموقع ذاكرة.

- The address lines are generally also used to address I/O ports.
- تُستخدم خطوط العناوين عمومًا أيضًا لمعالجة منافذ الإدخال/الإخراج.

- The higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module.
- تُستخدم البتات ذات الترتيب الأعلى لاختيار وحدة معينة على الناقل، بينما تختار البتات ذات الترتيب الأدنى موقع ذاكرة أو منفذ إدخال/إخراج داخل الوحدة.

- For example, on an 8-bit address bus, address 01111111 references locations in a memory module (module 0) with 128 words of memory. Address 10000000 and above refer to devices attached to an I/O module (module 1).
- على سبيل المثال، على ناقل عناوين مكون من ٨ بتات، يشير العنوان ٠١١١١١١١ إلى مواقع في وحدة ذاكرة (الوحدة ٠) تحتوي على ١٢٨ كلمة من الذاكرة. يشير العنوان ١٠٠٠٠٠٠٠ وما فوق إلى الأجهزة المرفقة بوحدة إدخال/إخراج (الوحدة ١).

Computer Design and Organization
تصميم وتنظيم الحاسوب

Instructions: :التعليمات
Language of the Computer
لغة الحاسوب

Content محتوى

- Operations and Operands of Computer Hardware
  - عمليات ومتغيرات أجهزة الكمبيوتر
- MIPS instruction تعليمات
  - Formats التنسيقات
  - Addressing العنونة
  - Instructions (arithmetic, logic, immediate, branching)
    - التعليمات (الحسابية، والمنطقية، والفورية، والتفرعية)
  - Instructions for Making Decisions , loop, procedure call
    - تعليمات اتخاذ القرار، والحلقة، واستدعاء الإجراء
- Representing Instructions in the Computer
  - تمثيل التعليمات في الكمبيوتر
- Signed and Unsigned Numbers
  - الأرقام الموقعة وغير الموقعة

Operations & operand of Computer Hardware

عمليات ومتغيرات أجهزة الكمبيوتر

Computer Languages: Machine Language

لغات الحاسوب: لغة الآلة

- To actually speak to electronic hardware, you need to send electrical signals (*on* and *off)*
  - *للتحدث فعليًا إلى الأجهزة الإلكترونية، تحتاج إلى إرسال إشارات كهربائية (تشغيل وإيقاف)*
- *So,* the computer alphabet is just two letters {0,1} called **binary digit (bit).**
  - **لذا، فإن أبجدية الكمبيوتر تتكون من حرفين فقط {٠,١} يسمى الرقم الثنائي (بت).**
- A command that computer hardware understands and obeys is called **instruction,** which is a collection of bits.
- يُطلق على الأمر الذي يفهمه جهاز الكمبيوتر ويطيعه اسم التعليمات، وهي عبارة عن مجموعة من البتات.
- A binary representation of machine instructions is called **machine language** (binary language understood by the machine)
  - يُطلق على التمثيل الثنائي لتعليمات الآلة اسم لغة الآلة (لغة ثنائية يفهمها الجهاز)
- Example: **1000110010100000**, which is instruction to add 2 numbers.
  - مثال: ١٠٠٠١١٠٠١٠١٠٠٠٠٠، وهي تعليمات لإضافة رقمين.
- Machine language was very hard and so tedious(مملة)
  - كانت لغة الآلة صعبة للغاية ومملة للغاية(مملة)

Computer Languages: Assembly Language

لغات الكمبيوتر: لغة التجميع

- **Assembly language is a** symbolic representation of machine instructions, which is closer to the way humans think.
  - لغة التجميع هي تمثيل رمزي لتعليمات الآلة، وهي أقرب إلى طريقة تفكير البشر.
  - Also called low level language (LLL)
    - وتسمى أيضًا لغة المستوى المنخفض (LLL)
- **Assembler** is a program used to translates a symbolic version of an instruction into the binary version.
  - المجمع هو برنامج يستخدم لترجمة نسخة رمزية من التعليمات إلى النسخة الثنائية.

Example, the programmer would write

add A,B → Assembler → 1000110010100000

This instruction is equivalent to A=A+B

- **Assembly language requires the programmer to write one line for every instruction that the computer will follow, forcing the programmer to think like the computer**.
  - تتطلب لغة التجميع من المبرمج كتابة سطر واحد لكل تعليمة يتبعها الكمبيوتر، مما يجبر المبرمج على التفكير مثل الكمبيوتر.

Computer Languages: High Level Language
لغات الكمبيوتر: لغة عالية المستوى

High-Level Language (HLL) is A portable programming language such as C, C++, Java, Visual Basic that is

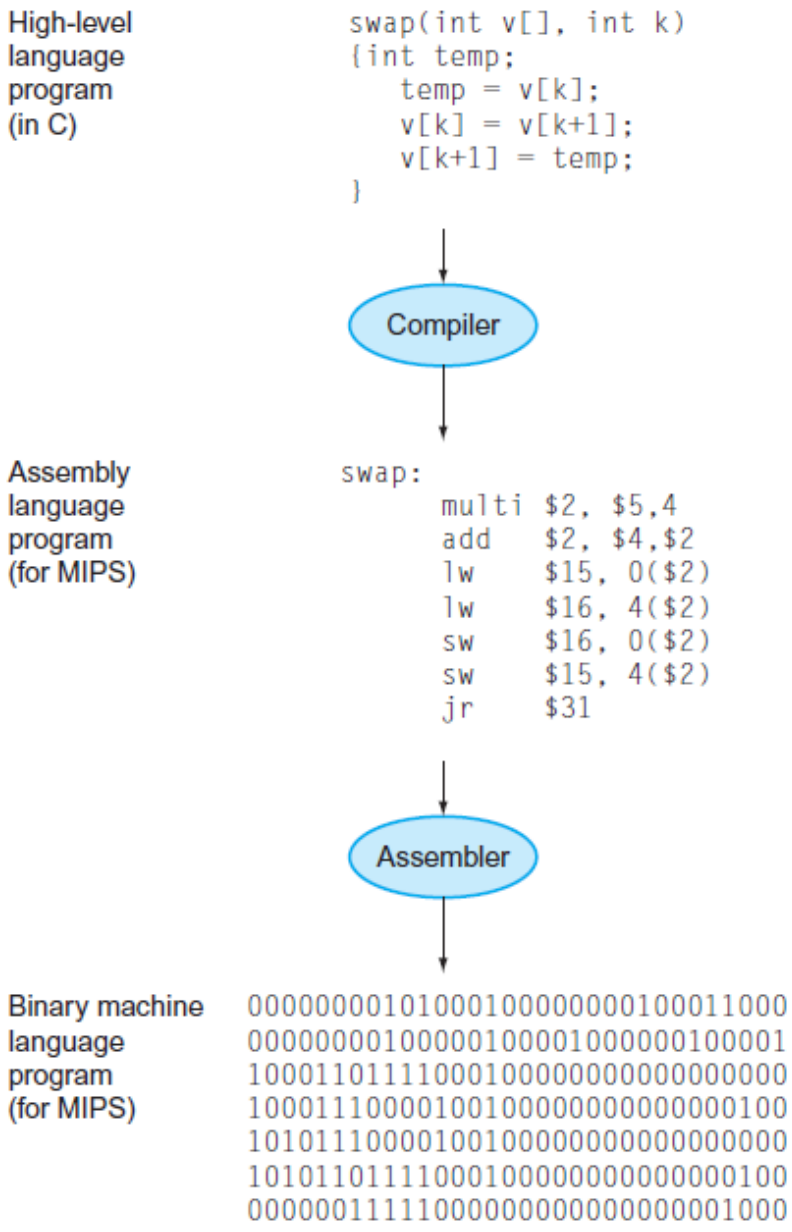
اللغة عالية المستوى (HLL) هي لغة برمجة محمولة مثل C و C++ و Java و Visual Basic وهي

- Composed of ***words*** and ***algebraic notation***
- ***تتكون من كلمات وتدوين جبري***
- Translated into assembly language by a ***compiler***.
- يتم ترجمتها إلى لغة التجميع بواسطة المترجم.

- **Compiler**: A program that translates high-level language statements into assembly language statements:
- المترجم: برنامج يترجم عبارات اللغة عالية المستوى إلى عبارات لغة التجميع:

Some compilers directly produce binary machine language

بعض المترجمين ينتجون لغة الآلة الثنائية مباشرة



Important benefits Of HLL فوائد هامة لـ

HLL offer several important benefits:

تقدم HLL العديد من الفوائد المهمة:

1) They allow the programmer to think in a more natural language.

٢) ١) تسمح للمبرمج بالتفكير بلغة أكثر طبيعية.

2) Designed domain-specific languages: design programming languages based on their use

٢) لغات مصممة لمجالات محددة: تصميم لغات برمجة بناءً على استخدامها

- Fortran was designed for scientific computation,
- تم تصميم Fortran للحوسبة العلمية،
- Cobol for business data processing,
- Cobol لمعالجة بيانات الأعمال،
- Lisp for symbol manipulation, etc.
- Lisp لمعالجة الرموز، إلخ.

3) Improving programmer productivity: Takes less time to develop programs when they are written in HLL

٣) تحسين إنتاجية المبرمج: يستغرق تطوير البرامج وقتًا أقل عند كتابتها بلغة HLL

- Require fewer lines to express an idea. تتطلب عددًا أقل من الأسطر للتعبير عن فكرة.
- Conciseness (الإيجاز) is an advantage of HLL over assembly language.
- الإيجاز (الإيجاز) هو ميزة لـ HLL مقارنة بلغة التجميع.

4) Allows programs to be independent of the computer on which they were developed, since compilers and assemblers can translate HLL programs to the target machine language.

٤) تسمح للبرامج بأن تكون مستقلة عن الكمبيوتر الذي تم تطويرها عليه، حيث يمكن للمترجمين والمجمعين ترجمة برامج HLL إلى لغة الآلة المستهدفة.

These advantages are so strong that today few programming is done in assembly language.

هذه المزايا قوية جدًا لدرجة أن القليل من البرمجة تتم اليوم بلغة التجميع.

Operations of Computer Hardware
عمليات أجهزة الكمبيوتر

Assembly Language

Instruction &Instruction set
التعليمات ومجموعة التعليمات

- The words of a computer's language are called **instructions**,
  - تُسمى كلمات لغة الكمبيوتر بالتعليمات،
- Its vocabulary is called an **instruction set**..وتسمى مفرداتها بمجموعة التعليمات
- A **Computer Program** is a collection of instructions that performs a specific task when executed by a computer.
  - برنامج الكمبيوتر هو مجموعة من التعليمات التي تؤدي مهمة محددة عند تنفيذها بواسطة الكمبيوتر.
- A **process** is program in execution (i.e. program loaded into memory (fully or partially) to be executed is called a process).
  - العملية هي برنامج قيد التنفيذ (أي البرنامج المحمل في الذاكرة (كليًا أو جزئيًا) ليتم تنفيذه يسمى عملية).
- **Stored-program concept**: Instructions and data of many types can be stored in memory as numbers.
  - مفهوم البرنامج المخزن: يمكن تخزين التعليمات والبيانات من أنواع عديدة في الذاكرة كأرقام.
- **Design *goal* of computer language هدف تصميم لغة الكمبيوتر**
  - To find a language that makes it *easy to build* the *hardware* and the *compiler* while maximizing *performance* and minimizing *cost*
    - إيجاد لغة تسهل بناء الأجهزة والمترجم مع تعظيم الأداء وتقليل التكلفة

How to Design the Instructions?
كيفية تصميم التعليمات؟

- Operations العمليات
  - Arithmetic الحسابية
  - Logical المنطقية
  - => Datapath
- Operands المتغيرات
  - => Datapath
- Control flow تدفق التحكم
  - Decision control التحكم في القرار
  - Procedures calls استدعاءات الإجراءات
  - => Control التحكم

Example

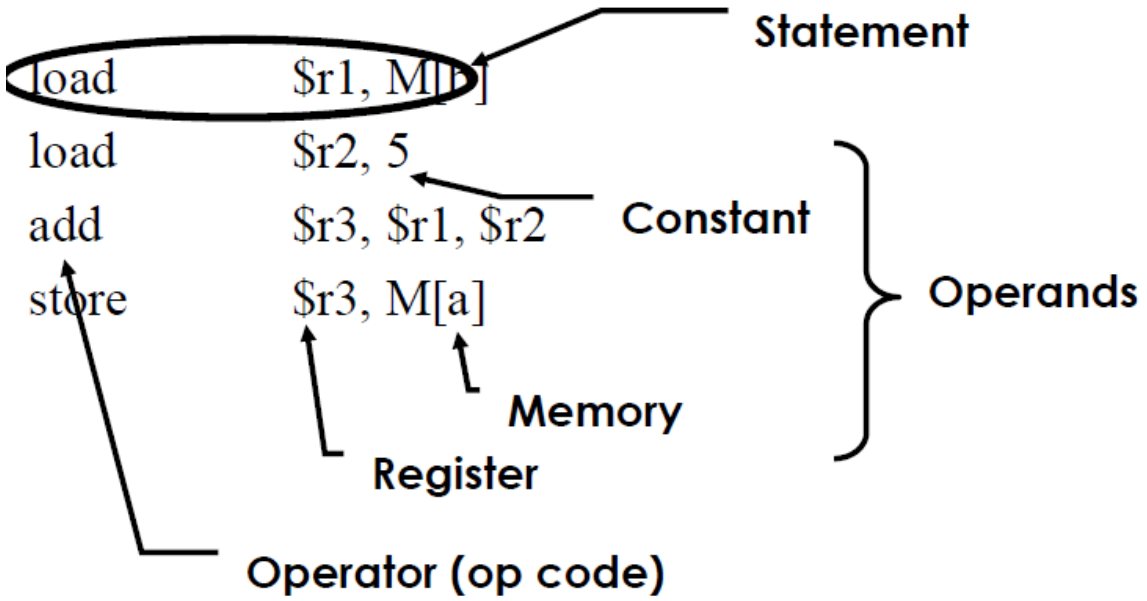
```
int add5 (int a)
{int tmp = a + 5; // Datapath= Arithmetic
return tmp;
}
void main ()
{ int a = 7; // Datapath
int c;
if (a == 7) // Decision control
c = add5(a); // Datapath =Procedures call
}
```

Recall C Language
استدعاء لغة C

- Operators: +, -, *, /, % (mod), ... المشغلات
- Operands: المتعاملات
  - Variables: sum, name1, x, etc. المتغيرات
  - Constants: 0, 1000, -17, 15.4 الثوابت
- Assignment statement: بيان الإسناد:
  - variable = expression المتغير = تعبير
- –Expressions consist of operators operating on operands,
  - –تتكون التعبيرات من مشغلات تعمل على المتعاملات،
  - Exp = 5*(salary-32)/9;
  - a = b+c+d-e;

C to Assembly Language (MIPS)
C إلى لغة التجميع (MIPS)

- a=b+5 ;



Design Principles مبادئ التصميم

- *Design Principle 1:* Simplicity favors regularity مبدأ التصميم ١: البساطة تفضل الانتظام
  - keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number.
    - الحفاظ على بساطة الأجهزة: الأجهزة لعدد متغير من المتغيرات أكثر تعقيدًا من الأجهزة لعدد ثابت.
- *Design Principle 2:* Smaller is faster. *مبدأ التصميم ٢: الأصغر أسرع.*
  - A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.
    - قد يؤدي عدد كبير جدًا من السجلات إلى زيادة وقت دورة الساعة ببساطة لأنه يستغرق إشارات إلكترونية وقتًا أطول عندما يتعين عليها السفر لمسافة أبعد.
  - The designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast.
    - يجب على المصمم موازنة رغبة البرامج في الحصول على المزيد من السجلات مع رغبة المصمم في الحفاظ على سرعة دورة الساعة.

What is MIPS?

- **MIPS** is one of the most popular processor architectures.
- **MIPS هي واحدة من أكثر بنيات المعالجات شيوعًا.**
- **MIPS** is an acronym for Microprocessor without Interlocked Pipeline Stages.
- MIPS هو اختصار لـ Microprocessor without Interlocked Pipeline Stages.
- The early **MIPS** architectures were 32-bit, with 64-bit versions added later.
- كانت بنيات MIPS المبكرة ٣٢ بت، مع إضافة إصدارات ٦٤ بت لاحقًا.

It involves: وهي تتضمن:

1. **MIPS Register File**
2. **Memory**
3. **MIPS Instructions**

1. MIPS Register File

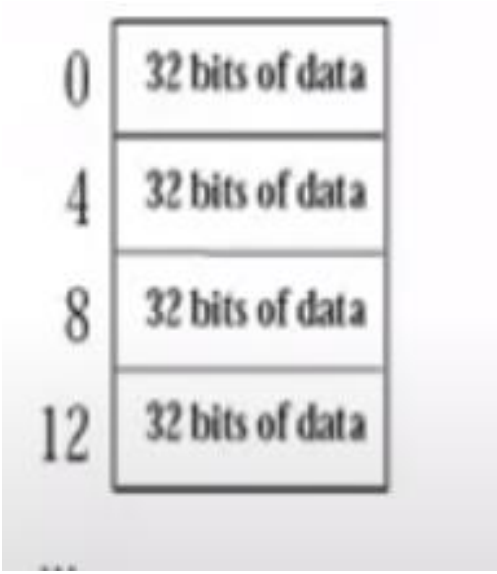
MIPS Register File:

- ✓ Register file is inside the CPU so accessing the registers in a register file is done quite quickly (fewer computer cycles).
- ✓ يوجد ملف السجل داخل وحدة المعالجة المركزية، لذا فإن الوصول إلى السجلات في ملف السجل يتم بسرعة كبيرة (عدد أقل من دورات الكمبيوتر).
- ✓ Each register stores the value of a variable.
- ✓ يخزن كل سجل قيمة متغير.
- ✓ Registers are represented with $, for example: $t0
- ✓ يتم تمثيل السجلات بالرمز $، على سبيل المثال: $t0

1. MIPS Register File

MIPS Register File:

- 32 bits * 32 units (only 32 registers provided and each register contains 32 bits (one word).
- ٣٢ بت * ٣٢ وحدة (يتم توفير ٣٢ سجلاً فقط ويحتوي كل سجل على ٣٢ بتًا (كلمة واحدة).



- For MIPS, a word is 32 bits or 4 bytes
- بالنسبة لـ MIPS، تكون الكلمة ٣٢ بتًا أو ٤ بايتات
- Registers hold 32 bits of data
- تحتوي السجلات على ٣٢ بتًا من البيانات

2. MIPS Memory

MIPS is a load-store architecture هي بنية تخزين تحميل

The memory associated with the MIPS architecture is **4GB**, and each memory location is 8 bits.

الذاكرة المرتبطة ببنية MIPSهي ٤ جيجابايت، وكل موقع ذاكرة هو ٨ بت.

In the register file each location is 32 bits which means its word addressable, however in MIPS memory each memory location is 8 bits only (1 byte)

في ملف السجل كل موقع هو ٣٢ بت مما يعني أنه يمكن عنونته بالكلمات، ومع ذلك في ذاكرة MIPSكل موقع ذاكرة هو ٨ بت فقط (١ بايت)

To address memory you need more clock cycles i.e. more time because memory is external, for example RAM is external and is not inside the CPU, so to access RAM we need more clock cycles.

لعنونة الذاكرة تحتاج إلى المزيد من دورات الساعة أي المزيد من الوقت لأن الذاكرة خارجية، على سبيل المثال ذاكرة الوصول العشوائي خارجية وليست داخل وحدة المعالجة المركزية، لذلك للوصول إلى ذاكرة الوصول العشوائي نحتاج إلى المزيد من دورات الساعة.

We always try to access the memory as less as possible. .نحاول دائمًا الوصول إلى الذاكرة بأقل قدر ممكن

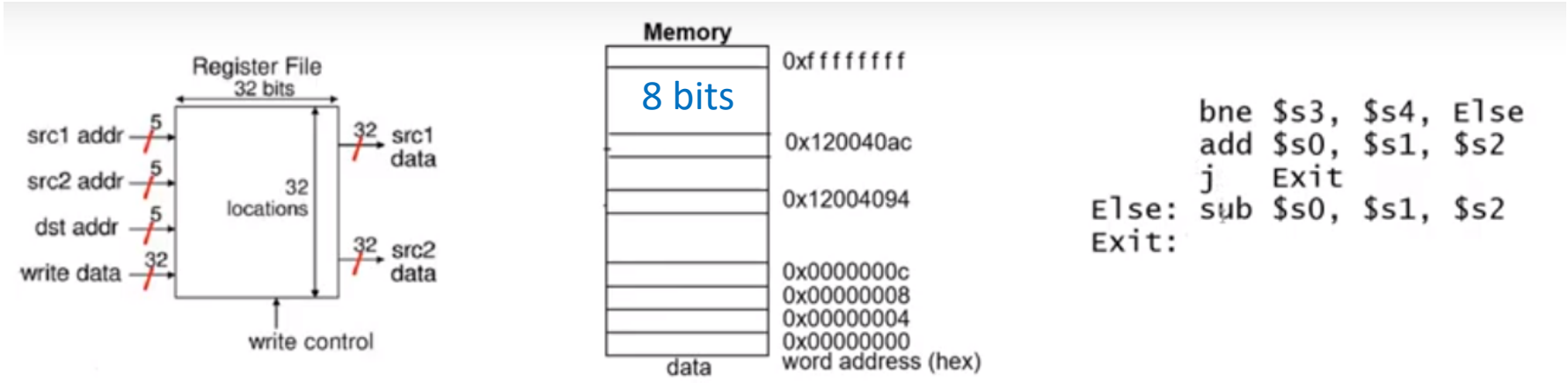
If you are going to access the memory for every operation this means you are going to take a very long time for the simplest operation, so the technique is to use the register file as much as possible and use the memory only when we want to load data from memory or store data into memory.

إذا كنت ستصل إلى الذاكرة لكل عملية فهذا يعني أنك ستستغرق وقتًا طويلاً لأبسط عملية، لذا فإن التقنية هي استخدام ملف السجل قدر الإمكان واستخدام الذاكرة فقط عندما نريد تحميل البيانات من الذاكرة أو تخزين البيانات في الذاكرة.

In MIPS architecture we are going to access the memory when we want to load data from memory into registers or to store data into memory from registers.

في بنية MIPSسنصل إلى الذاكرة عندما نريد تحميل البيانات من الذاكرة إلى السجلات أو تخزين البيانات في الذاكرة من السجلات.

MIPS



Every single space in the memory is 8 bits. Now the problem is that in each register in the register file the size is 32 bits. How are we going to bring data from memory into the register and registers into the memory?

كل مساحة مفردة في الذاكرة هي ٨ بت. الآن المشكلة هي أن حجم كل سجل في ملف السجل هو ٣٢ بت. كيف سنجلب البيانات من الذاكرة إلى السجل والسجلات إلى الذاكرة؟

The solution is to access the memory in groups of 4, so four of memory locations will be used to store a single data from a register, i.e. four memory locations make up one register location.

الحل هو الوصول إلى الذاكرة في مجموعات من ٤، لذلك سيتم استخدام أربعة من مواقع الذاكرة لتخزين بيانات واحدة من سجل، أي أن أربعة مواقع ذاكرة تشكل موقع سجل واحد.

- *The addresses beside the memory are word address (32 bits), in other words the data size inside the memory is 8 bits but we need 32 bits to address each location.*
- *العناوين بجانب الذاكرة هي عناوين كلمات (٣٢ بت)، بمعنى آخر حجم البيانات داخل الذاكرة هو ٨ بت ولكننا نحتاج إلى ٣٢ بت لمعالجة كل موقع.*

3. MIPS Instructions

MIPS instructions help the transfer of data between the register file and the memory.

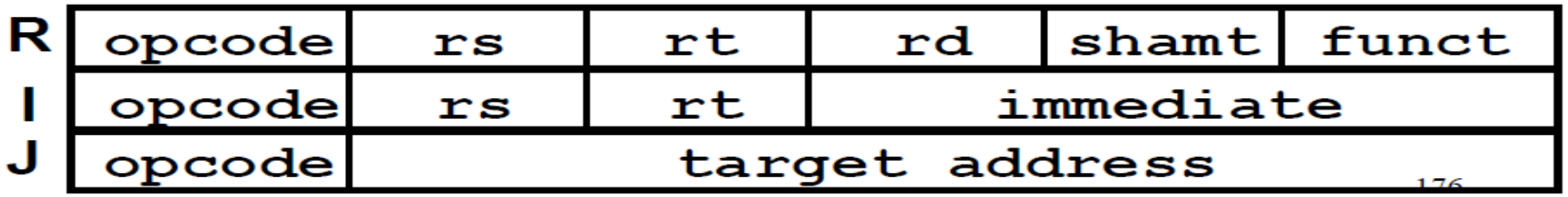
تساعد تعليمات MIPS في نقل البيانات بين ملف السجل والذاكرة.

MIPS Instruction Set Architecture – (MIPS ISA)

MIPS – (MIPS ISA) بنية مجموعة تعليمات

Instruction Categories فئات التعليمات

- Computational (add, sub) الحوسبة (إضافة، فرع)
- Branch - Load/Store الفرع - التحميل/التخزين
- Jump الانتقال
- Memory Management إدارة الذاكرة
- Special خاص



3 Instruction Formats: all 32 bits wide ٣ تنسيقات للتعليمات: جميعها بعرض ٣٢ بت

Types of Operands in MIPS

- 3 Types operands ٣ أنواع من المتغيرات

1. Register operands : All arithmetic operations are in the register operands

١. متغيرات السجل: جميع العمليات الحسابية موجودة في متغيرات السجل

MIPS operands

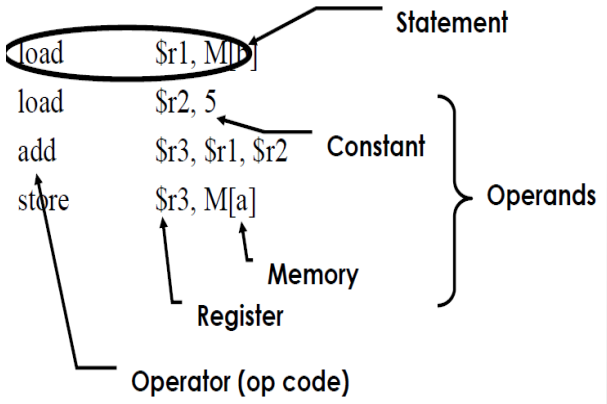
Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register $zero always equals 0, and register $at is reserved by the assembler to handle large constants.
2^{32} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

2. Memory operands: Array or structure ٢. متغيرات الذاكرة: المصفوفة أو البنية

- Only *load/store* can access memory
- يمكن فقط لـ load/store الوصول إلى الذاكرة

3. Constant or immediate operands ٣. متغيرات ثابتة أو فورية

- Small value will be in the instruction
- ستكون القيمة الصغيرة موجودة في التعليمات
- Large value will be stored separately
- ستُخزن القيمة الكبيرة بشكل منفصل



MIPS Instructions

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Operands and Registers المتعاملات و السجلات

- Unlike high-level language, MIPS assembly don't use variables
  - على عكس لغة المستوى العالي، لا تستخدم لغة تجميع MIPS المتغيرات
- Assembly operands : registers متغيرات التجميع: السجلات
  - Limited number of special locations built directly into the hardware
    - عدد محدود من المواقع الخاصة المضمنة مباشرة في الأجهزة
- Benefits: :الفوائد
  - Registers in hardware => faster than memory السجلات في الأجهزة => أسرع من الذاكرة
  - Registers are easier for a compiler to use e.g., as a place for temporary storage
    - السجلات أسهل على المترجم في الاستخدام، على سبيل المثال، كمكان للتخزين المؤقت
- Registers can hold variables to يمكن للسجلات الاحتفاظ بالمتغيرات
  - *reduce memory traffic and لتقليل حركة الذاكرة و*
  - *improve code density* (register named with fewer bits than memory location)
    - تحسين كثافة التعليمات البرمجية (سجل يسمى بعدد بتات أقل من موقع الذاكرة)
- The operation of moving a variable from a register to memory is called ***spilling***, (store)
  - تسمى عملية نقل متغير من سجل إلى الذاكرة بالانسكاب (التخزين)
- while the reverse operation of moving a variable from memory to a register is called ***filling*** (load).
  - بينما تسمى العملية العكسية لنقل متغير من الذاكرة إلى سجل بالملء (التحميل).
- ***Such a variable has a much slower processing speed than a variable in a register.***
  - ***يتمتع مثل هذا المتغير بسرعة معالجة أبطأ بكثير من المتغير الموجود في سجل.***

Role of Registers vs. Memory
دور السجلات مقابل الذاكرة

- What if more variables than registers?
- ماذا لو كان عدد المتغيرات أكبر من عدد السجلات؟
  - Compiler tries to keep most frequently used variables in registers
  - يحاول المترجم الاحتفاظ بالمتغيرات الأكثر استخدامًا في السجلات
  - Writes less common variables to memory
  - يكتب متغيرات أقل شيوعًا في الذاكرة
- Why not keep all variables in memory?
- لماذا لا يحتفظ بجميع المتغيرات في الذاكرة؟
  - Smaller is faster: registers are faster than memory
  - الأصغر أسرع: السجلات أسرع من الذاكرة
  - Registers more versatile(متعدد الاستخدام): :(متعدد الاستخدام) السجلات أكثر تنوعًا
    - MIPS arithmetic instructions can read 2 registers, operate on them, and write 1 per instruction
    - يمكن لتعليمات MIPS الحسابية قراءة سجلين والعمل عليهما وكتابة سجل واحد لكل تعليمة
    - MIPS data transfers only read or write 1 operand per instruction, and no operation
    - تنتقل بيانات MIPS فقط لقراءة أو كتابة متغير واحد لكل تعليمة، ولا توجد عملية

MIPS Registers

- MIPS is a load-store architecture,
- MIPSعبارة عن بنية تحميل وتخزين،
  - only load and store instructions can access memory.
  - فقط تعليمات التحميل والتخزين يمكنها الوصول إلى الذاكرة.
  - All other instructions (add, sub, mul, div, and, or, etc.) must get their operands from registers and store their results in a register.
  - يجب أن تحصل جميع التعليمات الأخرى (add، sub، mul، div، and، or، إلخ) على متغيراتها من السجلات وتخزين نتائجها في سجل.
- 32 registers, each is 32 bits wide (called word in MIPS).
- ٣٢ سجلاً، كل منها بعرض ٣٢ بت (تسمى كلمة في MIPS).
- Why 32 register? smaller is faster لماذا ٣٢ سجلاً؟ كلما كان أصغر كان أسرع
- Registers are numbered from 0 to 31 يتم ترقيم السجلات من ٠ إلى ٣١
- Each can be referred to by number or name يمكن الإشارة إلى كل منها برقم أو اسم
  - $0, $1, $2, … $30, $31

By convention, each register also has a name to make it easier to code, e.g., $16 - $23 → $s0 - $s7 (C variables)

وفقًا للاتفاقية، يكون لكل سجل أيضًا اسم لتسهيل الترميز، على سبيل المثال، $١٦ - $٢٣ □ $ $s7 - $s0(متغيرات C)

- $8 - $15 → $t0 - $t7 (temporary)
- Others: HI, LO, PC

MIPS Registers Cont.

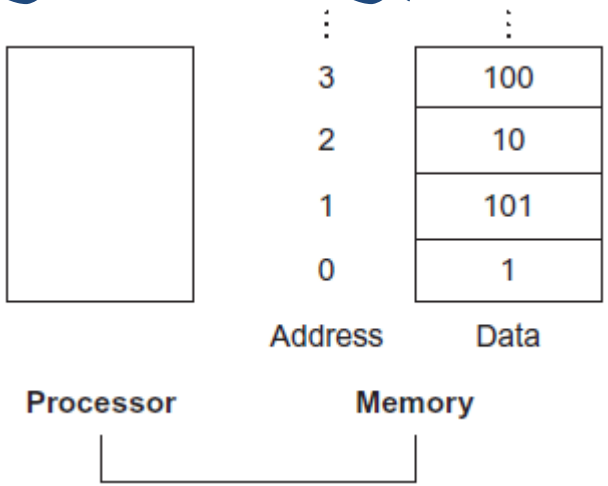
Name	Register Number	Usage	Should preserve on call?
$zero	0	the constant 0	no
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

To compile a proced
compiler must kno
registers nee
to be preserved and w
modified without

Register 1 ($at) reserved for assembler, 26-27 for operating system

Memory Operands متغيرات الذاكرة

Programming languages have تحتوي لغات البرمجة على
- **Simple variables** that contain single data elements, متغيرات بسيطة تحتوي على عناصر بيانات مفردة،
- **Arrays and structures**: more complex data structures المصفوفات والهياكل: تحتوي هياكل البيانات الأكثر تعقيدًا
  - contain many more data elements than there are registers in a computer على عناصر بيانات أكثر بكثير من عدد السجلات في الكمبيوتر



Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0

الذاكرة عبارة عن مصفوفة كبيرة أحادية البعد، حيث يعمل العنوان كمؤشر لتلك المصفوفة، بدءًا من ٠

- The processor can keep only a small amount of data in registers, لا يمكن للمعالج الاحتفاظ إلا بكمية صغيرة من البيانات في السجلات،
- but computer memory contains billions of data elements. ولكن ذاكرة الكمبيوتر تحتوي على مليارات من عناصر البيانات.
- Hence, data structures (arrays and structures) are kept in memory. وبالتالي، يتم الاحتفاظ بهياكل البيانات (المصفوفات والهياكل) في الذاكرة.
- MIPS must include instructions that transfer data between memory and registers. يجب أن تتضمن MIPS تعليمات تنقل البيانات بين الذاكرة والسجلات.

called **data transfer instructions**. تسمى تعليمات نقل البيانات.

- ***load*** *:* data transfer instruction that copies data from memory to a register
  - load : تعليمات نقل البيانات التي تنسخ البيانات من الذاكرة إلى سجل
- ***Store***: data transfer instruction that copies data to memory from a register
  - Store : تعليمات نقل البيانات التي تنسخ البيانات إلى الذاكرة من سجل

- To access a word in memory, the instruction must supply the memory **address**. للوصول إلى كلمة في الذاكرة، يجب أن توفر التعليمات عنوان الذاكرة.

Clock cycle & Instruction code
دورة الساعة ورمز التعليمات

- A CPU register can generally be accessed in **a single clock cycle**,
- يمكن الوصول إلى سجل وحدة المعالجة المركزية بشكل عام في دورة ساعة واحدة،
- Main memory may require number of CPU clock cycles to read or write.
- قد تتطلب الذاكرة الرئيسية عددًا من دورات ساعة وحدة المعالجة المركزية للقراءة أو الكتابة.
- Since there are very few registers compared to memory cells, registers also require far fewer bits to specify which register to use.
- نظرًا لوجود عدد قليل جدًا من السجلات مقارنة بخلايا الذاكرة، تتطلب السجلات أيضًا عددًا أقل بكثير من البتات لتحديد السجل الذي يجب استخدامه.

This in turn allows for smaller instruction codes.

هذا بدوره يسمح برموز تعليمات أصغر.

- In MIPS processor , there is 32 general-purpose registers, so it takes 5 bits to specify which one to use.
- في معالج MIPS، يوجد ٣٢ سجلًا للأغراض العامة، لذا يستغرق الأمر ٥ بتات لتحديد أي منها يجب استخدامه.
- In contrast, the MIPS has a 4-gigabyte memory capacity, so it takes 32 bits to specify which memory cell to use.
- على النقيض من ذلك، يتمتع معالج MIPSبسعة ذاكرة ٤ جيجابايت، لذا يستغرق الأمر ٣٢ بتًا لتحديد خلية الذاكرة التي يجب استخدامها.
- So, the instruction code of instruction with 3 operands will require
- لذا، فإن رمز التعليمات الخاص بالتعليمات التي تحتوي على ٣ متغيرات سيتطلب
  - 15 bits if they are all registers, and ١٥ بتًا إذا كانت جميعها سجلات، و
  - 96 bits if they are all memory addresses. ٩٦ بتًا إذا كانت جميعها عناوين ذاكرة.

HW/SW IF: How Compiler Use Registers
HW/SW IF: كيفية استخدام المترجم للسجلات

- Problem: more variables than available registers
  - المشكلة: عدد المتغيرات أكبر من عدد السجلات المتاحة
- Solution الحل
  - Keep *the most frequently used variables* in registers
    - احتفظ بالمتغيرات الأكثر استخدامًا في السجلات
  - Place the rest in memory (called *spilling registers*)
    - ضع الباقي في الذاكرة (وتسمى سجلات الإراقة)
  - ✓ Use load and store to move variables between registers and memory
    - ✓ استخدم التحميل والتخزين لنقل المتغيرات بين السجلات والذاكرة
- Why? لماذا؟
  - Register is faster but its size is small السجل أسرع ولكن حجمه صغير
  - Compiler must use register efficiently يجب على المترجم استخدام السجل بكفاءة
- How to compile the following C statement to MIPS?
  - كيف يمكن تجميع عبارة C التالية إلى MIPS؟

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

- Assume f, g, h, i, j uses $s0, .. $s4

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

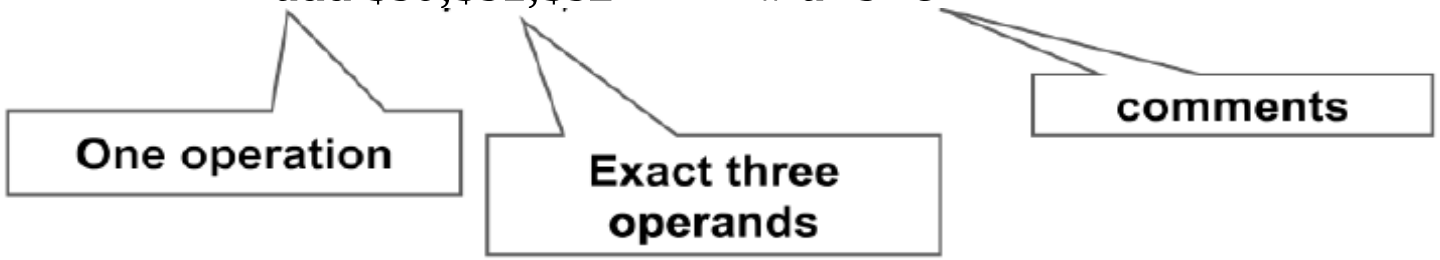
Operations of the Computer Hardwar: Arithmetic Operations
عمليات أجهزة الحاسوب: العمليات الحسابية

- All arithmetic instructions have 3 operands تحتوي جميع التعليمات الحسابية على ٣ متغيرات
  - Operand order is fixed (destination first) ترتيب المتغيرات ثابت (الوجهة أولاً)
- Ex; Convert the following C code to MIPS
  - مثال؛ تحويل الكود Cالتالي إلى MIPS

```
a = b + c;
d = a – e;
```

- A MIPS instruction operates on **two source operands** and places the result in **one destination operand**
  - **تعمل تعليمات MIPSعلى متغيرين مصدرين وتضع النتيجة في متغير وجهة واحد**
- Assume a,b, c, e, d uses registers $s0 to $s4 respectively
  - افترض أن a,b,c,e,dتستخدم السجلات $ s0إلى $ s4على التوالي

```
add $s0,$s1,$s2        # a=b+c
```




```
sub $s4, $s0, $s3      # d=a-e
```

Compiling an Assignment when an Operand is in Memory

Ex: **Compile this C assignment statement: g = h + A[8];**

على سبيل المثال: قم بتجميع بيان التعيين C هذا: g = h + A[8];

Let **A is an array of 100** words

دع A عبارة عن مصفوفة مكونة من ١٠٠ كلمة

Assume variables **g** in register **$s1**, and **h** in register **$s2** .

افترض أن المتغيرات g في السجل $s1، و h في السجل $s2.

Assume that the starting address (***base address***), *of the array is in* **$s3**.

افترض أن عنوان البداية (عنوان القاعدة) للمصفوفة موجود في $s3.

Code compiled to MIPS: الكود المترجم إلى

S3 is base register
8 is offset

```
lw $t0,8($s3)      # Temporary reg $t0 gets A[8]
add $s1,$s2,$t0    # g = h + A[8]
```

lw : is load instruction هي تعليمات تحميل

- In this assignment statement, one of the operands is in memory, so we must first transfer A[8] to a register.
- في بيان التعيين هذا، يوجد أحد المتغيرات في الذاكرة، لذا يجب علينا أولاً نقل A[8] إلى سجل.
- The address of this array element (**lw $t0,8($s3))**
- عنوان عنصر المصفوفة هذا (lw $t0,8($s3))

= ***base address*** (in $s3 called *base register*)+***array index or offset*** (8).

= عنوان القاعدة (في $s3 يسمى السجل الأساسي)+فهرس المصفوفة أو الإزاحة (٨).

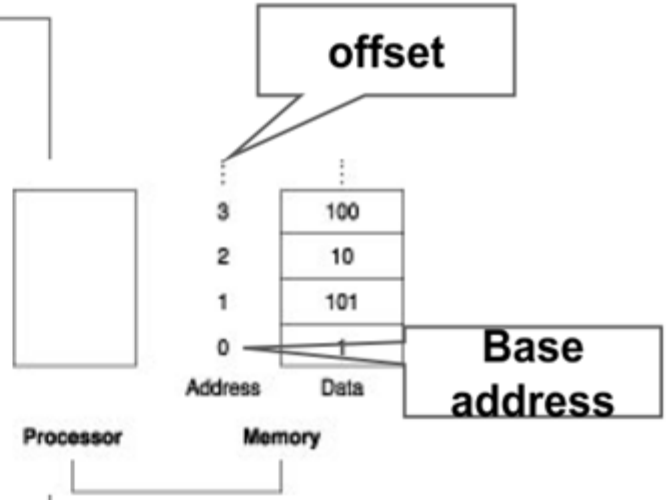
The data should be placed in a temporary register (**$t0**)for use in the next instruction.

يجب وضع البيانات في سجل مؤقت ($t0) لاستخدامها في التعليمات التالية.

Array Example مثال على المصفوفة

- **Load format تنسيق التحميل**
  - **lw register names, const offset(base register)**
  - **أسماء السجلات lw، إزاحة ثابتة (السجل الأساسي)**

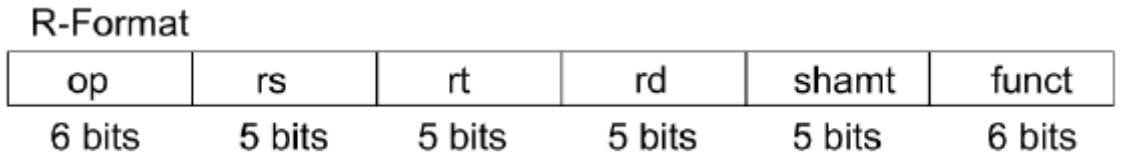
```
g = h + a[8]
assume g, h => $s1, $s2
base address => $s3

lw $t0, 8($s3)  #lw: load word
add $s1, $s2 , $t0

If a[12] = h + a[8]
Add one more instructions
sw $s1,12[$s3]
```



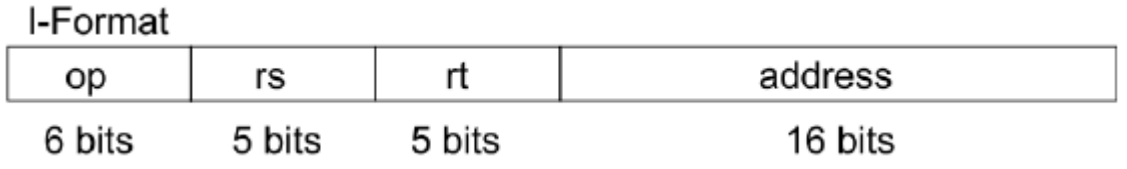
Offset (*in bytes*) from this pointer 8($s3) means 8 bytes added to content of $S3
الإزاحة (بالبايت) من هذا المؤشر ٨($s3) تعني إضافة ٨ بايتات إلى محتوى $S3

MIPS Instruction Format

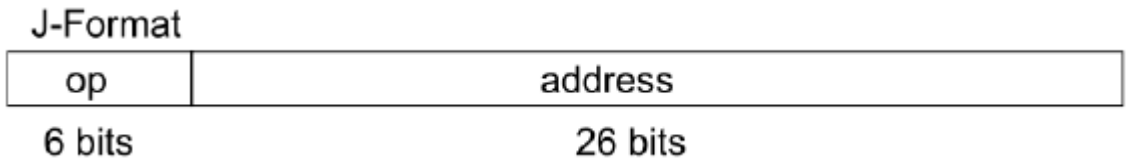
- One instruction is 32 bits تتكون التعليمات الواحدة من ٣٢ بت
  - Divide instruction word into "*fields*" "*تقسيم كلمة التعليمات إلى "حقول*"
  - Each field tells computer something about instruction
    - يخبر كل حقل الكمبيوتر بشيء ما عن التعليمات
- We could define different fields for each instruction, but MIPS is based on simplicity, so
  - يمكننا تعريف حقول مختلفة لكل تعليمة، ولكن MIPS يعتمد على البساطة، لذا
- define 3 basic types of instruction formats:
  - قم بتحديد ٣ أنواع أساسية من تنسيقات التعليمات:
  - *R-format: for register*



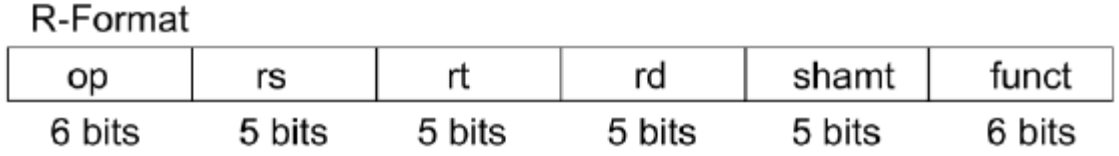
  - *I-format: for immediate, and **lw** and **sw** (since the* offset counts as an immediate)



  - *J-format: for jump*



R-Format Instructions



- opcode: *operation of instruction (Note: 0 for all R-Format instructions)*
- *opcode: تشغيل التعليمات (ملاحظة: ٠ لجميع تعليمات تنسيق R)*
- rs (Source Register): *generally used to specify register containing first* operand
- rs(سجل المصدر): يستخدم بشكل عام لتحديد السجل الذي يحتوي على المتغير الأول
- rt (target register/second source): *generally used to specify register containing second* operand
- rt(سجل الهدف/المصدر الثاني): يستخدم بشكل عام لتحديد السجل الذي يحتوي على المتغير الثاني
- rd (Destination Register): *generally used to specify register which will* receive result of computation
- rd(سجل الوجهة): يستخدم بشكل عام لتحديد السجل الذي سيستقبل نتيجة الحساب
- shamt: shift amount مقدار التحويل
- funct: function; this field selects the *variant of the operation in the op field* called *function code*
- *funct: دالة؛ يحدد هذا الحقل متغير العملية في حقل op المسمى كود الدالة*
- Question: Why aren't opcode and funct a single 12-bit field?
- السؤال: لماذا لا يكون opcode وfunct حقلًا واحدًا مكونًا من ١٢ بت؟

R-Format Instructions

- If we combine *opcode* and *funct* for R instruction, the instruction format is not so consistent between R and I, which may make processor's design complex.
- إذا قمنا بدمج التعليمات البرمجية والدالة لتعليمات R، فإن تنسيق التعليمات ليس متسقًا جدًا بين Rوا، مما قد يجعل تصميم المعالج معقدًا.

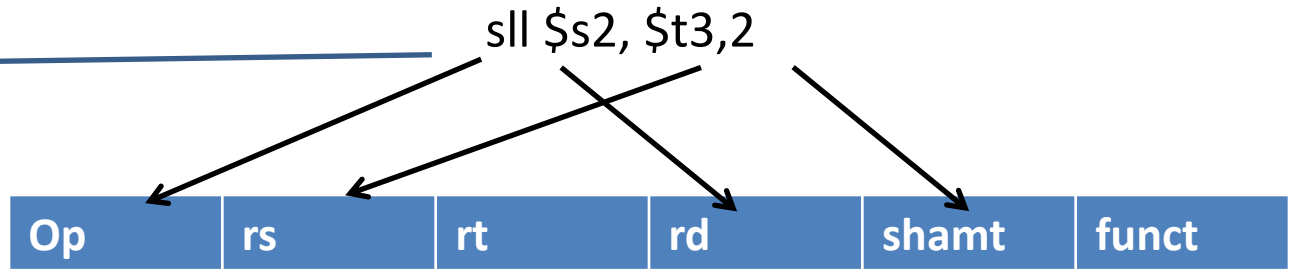
- Since the *opcode* is the same for some operation in MIPS and if you change the *funct* then you can't differentiate which operation the instruction does, for example consider the following add(R,0,32): **ADD** has *opcode* 0 and *funct* 32; consider that and(R,0,36) and has also opcode 0 but different *funct* in this case 36 which means it's an **AND** operation.
- نظرًا لأن التعليمات البرمجية هي نفسها لبعض العمليات في MIPSوإذا قمت بتغيير الدالة، فلن تتمكن من التمييز بين العملية التي تقوم بها التعليمات، على سبيل المثال، ضع في اعتبارك ما يلي: add(R,0,32)تحتوي علىADD التعليمات البرمجية ٠ والدالة ٣٢؛ ضع في اعتبارك أن and(R,0,36)وتحتوي أيضًا على التعليمات البرمجية ٠ ولكن دالة مختلفة في هذه الحالة ٣٦ مما يعني أنها عملية AND.

R-Format Instructions (Cont.)

- Notes about register fields: ملاحظات حول حقول السجل:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31.
  - يتكون كل حقل سجل من ٥ بتات بالضبط، مما يعني أنه يمكنه تحديد أي عدد صحيح غير موقّع في النطاق من ٠ إلى ٣١.
  - Each of these fields specifies one of the 32 registers by number.
  - يحدد كل من هذه الحقول أحد السجلات الـ ٣٢ بالرقم.
- Final field: الحقل النهائي:
- shamt: contains the amount a shift instruction will shift by.
  - shamt: يحتوي على المقدار الذي سيتم تحويله بواسطة تعليمة التحويل.
  - Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
  - إن تحويل كلمة مكونة من ٣٢ بتًا بأكثر من ٣١ أمر غير مجدٍ، لذا فإن هذا الحقل يتكون من ٥ بتات فقط
  - This field is set to 0 in all but the shift instructions
  - يتم تعيين هذا الحقل على ٠ في جميع التعليمات باستثناء تعليمات التحويل

MIPS R Type Instruction

Shift left sll
Shift right srl



Shift left. $s2 is the destination. $t3 is the source,

تحويل إلى اليسار. $ s2هو الوجهة. $ t3هو المصدر،

2 is the shift amount (the number of bits to shift)

٢ هو مقدار التحويل (عدد البتات المراد تحويلها)

The content inside the register $t3 is going to be shifted lift by two bits and the result is going to be stored in register $s2

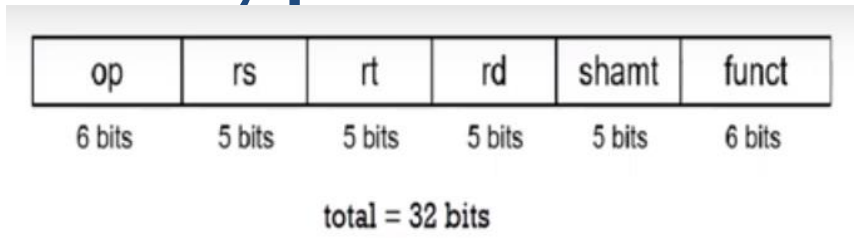
سيتم تحويل المحتوى داخل السجل $ t3بمقدار بت واحد وستُخزن النتيجة في السجل $s2

Examples of other R type instruction: mult, nor, xor, or,,srl etc.

أمثلة على تعليمات أخرى من نوع R: mult، nor، xor، or،srl,وما إلى ذلك.

MIPS R Type Instruction

	Should preserve on call?
	no
es	**no**
	yes
	no
	yes
	no
	yes
	yes
	yes
s	yes



Note:
$t0 = $8
$s1 = $17
$s2= $18
funct add= 32

1. add $t0, $s1, $s2

op: operand for R type instruction 000000 (6 bits)

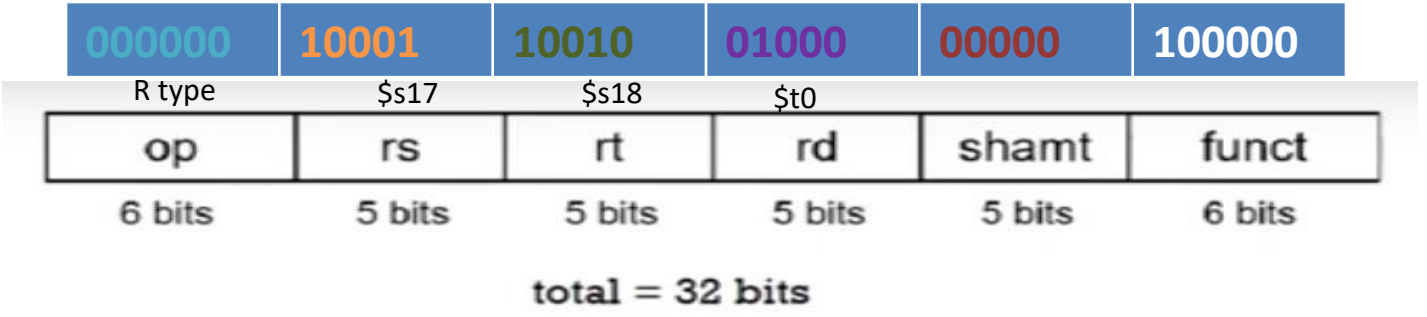
rs: $s1=$17 = 10001 (5 bits)

rt: $s2= = $18= $10010 (5 bits)

rd: $t0= $8= 01000 (5 bits)

shamt: if there is no shift value, shamt = 00000

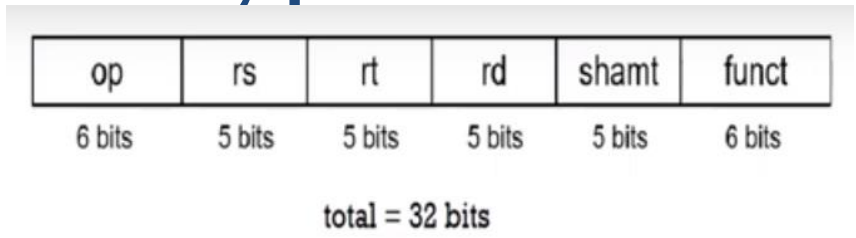
funct: add = 32 (dec) = 100000 (given)



Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
		op	rs	rt	rd	shamt	funct	
add	R	0	2	3	1	0	32	add $1, $2, $3
addu	R	0	2	3	1	0	33	addu $1, $2, $3
sub	R	0	2	3	1	0	34	sub $1, $2, $3
subu	R	0	2	3	1	0	35	subu $1, $2, $3
and	R	0	2	3	1	0	36	and $1, $2, $3
or	R	0	2	3	1	0	37	or $1, $2, $3
nor	R	0	2	3	1	0	39	nor $1, $2, $3
slt	R	0	2	3	1	0	42	slt $1, $2, $3
sltu	R	0	2	3	1	0	43	sltu $1, $2, $3
sll	R	0	0	2	1	10	0	sll $1, $2, 10
srl	R	0	0	2	1	10	2	srl $1, $2, 10

Name	Register Number	Usage	Should preserve on call?
$zero	0	the constant 0	no
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

MIPS R Type Instruction



Note:
$s2= $18
$t3 = $11
sll= 000000

ge	Should preserve on call?
ant 0	no
values	no
s	yes
es	no
ues	yes
es	no
nter	yes
nter	yes
nter	yes
dress	yes

2. sll $s2, $t3,2

op: operand for R type instruction 000000 (6 bits)

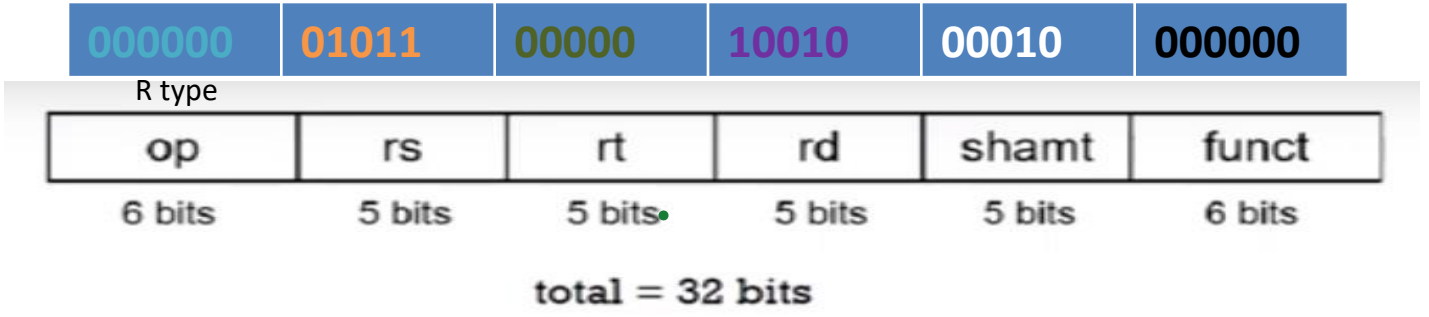
rs: $t3 = 01011 (5 bits) ($11)

rt: = 00000 (5 bits) no second source

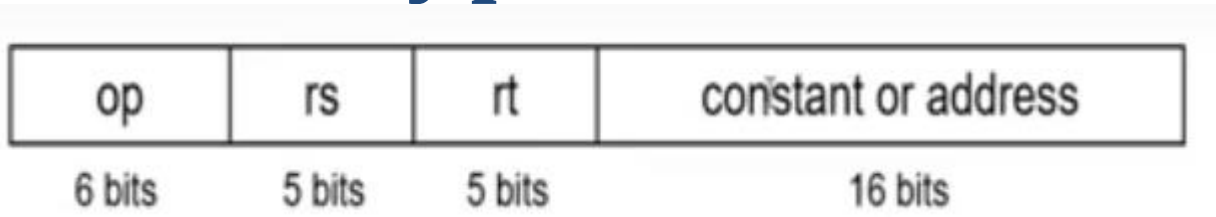
rd: $s2= 10010 (5 bits) ($18)

shamt: 00010 (binary representation of 2)

funct: sll = xxxxxx (given in the question)



MIPS I-Type Instruction



Examples:

1. lw $t0, 4($s3)
2. sw $t0, 8($s3)
3. addi $t0, $t1, 14 //R-type: add $t0, $s0, $s1
4. beq $t0, $t1, else

op - 6 bits opcode that specifies the operation

rs - 5 bits register file address of the first source operand

rt - 5 bits register file address of the second source/ result's destination

Constant or address – 16 bits (*the* offset counts as an immediate)

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
		op	rs	rt	immediate			
beq	I	4	1	2	25 (offset)			beq $1, $2, 100
bne	I	5	1	2	25 (offset)			bne $1, $2, 100
addi	I	8	2	1	100			addi $1, $2, 100
addiu	I	9	2	1	100			addiu $1, $2, 100
andi	I	12	2	1	100			andi $1, $2, 100
ori	I	13	2	1	100			ori $1, $2, 100
slti	I	10	2	1	100			slti $1, $2, 100
sltiu	I	11	2	1	100			sltiu $1, $2, 100
lui	I	15	0	1	100			lui $1, 100
lw	I	35	2	1	100 (offset)			lw $1, 100($2)
sw	I	43	2	1	100 (offset)			sw $1, 100($2)

MIPS I-Type Instruction

	Should preserve on call?
	no
s	no
	yes
	no
	yes
	no
	yes
	yes
	yes
s	yes

I-Format

op	rs	rt	Immediate / address / offset
6 bits	5 bits	5 bits	16 bits

1. **lw $t0, 4($s3)** → this means move the data from memory address 4($s3) into the register $t0.

4($s3) **means** memory address of ((4 + address of $s3))

op code: 35 decimal = 100011 binary

rs (source): ($s3) ↔ $19 (register 19) → 10011

rt (destination): $t0 ↔ $8 (register 8) → 01000

constant : 4 (decimal) → 100 (binary)

op (6 bits)	rs (5 bits)	rt (5 bits)	constant (16 bits)
100011	10011	01000	0000 0000 0000 0100
lw	$19	$8	4
Load	$s3	$t0	offset

Total : 32 bits

	Should preserve on call?
	no
s	**no**
	yes
	no
	yes
	no
	yes
	yes
	yes
s	yes

MIPS I-Type Instruction

I-Format

2. **sw $t0, 8($s3)** → this means move the data from the register $t0 into memory address 8($s3).

8($s3) **means** memory address of ((8 + address of $s3))

op code: 43 decimal = 101011 binary

rs (source): $t0 ↔ $8 (register 8) → 01000

rt (destination): ($s3) ↔ $19 (register 19) → 10011

constant : 8 → 1000

In the lw, the first reg. is the destination

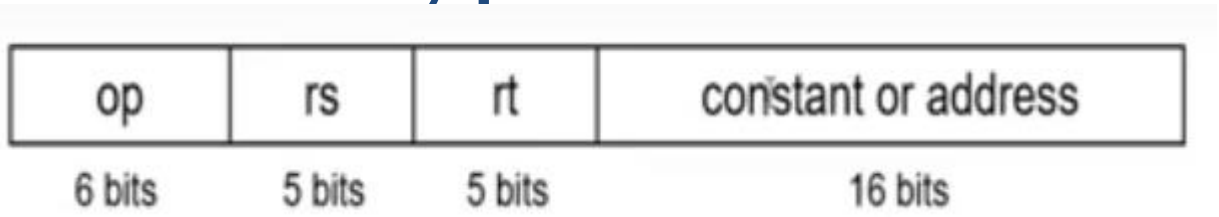
In the sw, the first reg. is the source

op (6 bits)	rs (5 bits)	rt (5 bits)	constant (16 bits)
101011	01000	10011	0000 0000 0000 1000
sw	$8	$19	8

Total : 32 bits

	Should preserve on call?
	no
es	**no**
	yes
	no
	yes
	no
	yes
	yes
	yes
s	yes

MIPS I-Type Instruction



3. **addi $t0, $t1, 14** → this means add 14 to the data in register $t1 and store the result in register $t0

- Any value that is outside the register is called immediate, and when you're adding or doing any operation between a register and an immediate value.

op code: 8 decimal = 001000 binary

rs (source): $t1 ↔ $9 (register 9) → 01001

rt (destination): ($t0) ↔ $8 (register 8) → 01000

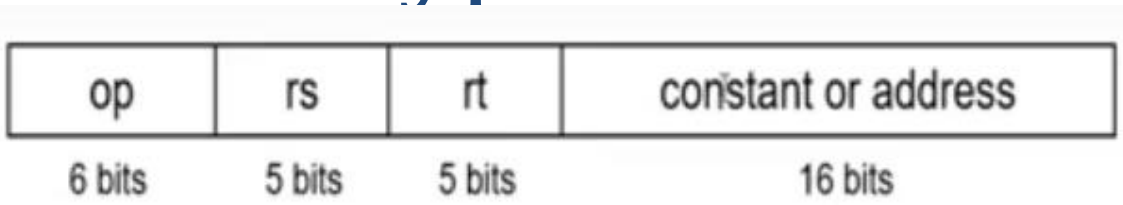
constant : 14 → 1110

op (6 bits)	rs (5 bits)	rt (5 bits)	constant (16 bits)
001000	01001	01000	0000 0000 0000 1110
addi	$9	$8	14

Total : 32 bits

MIPS I-Type Instruction

	Should preserve on call?
	no
es	**no**
	yes
	no
	yes
	no
	yes
	yes
	yes
s	yes



4. beq $t0, $t1, 14 → compares the value of $t0 and $t1 and if they are equal the code will go to the else's address (i.e. 14).

- 14 represents the else variable address
- ➢ beq means branch if equal to
- ➢ beq is equivalent to if/else (if $t0 == $t1) go to else's address
- ➢ There is no destination, we are just comparing two registers and if they're equal the code will jump to else address

op code: 4 decimal = 000100 binary

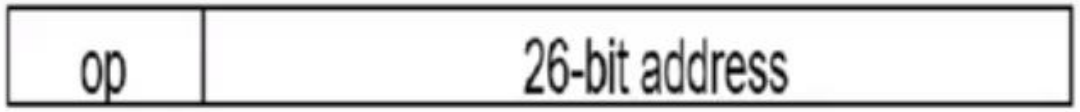
rs (source): $t0 ↔ $8 (register 8) → 01000

rt (source): ($t1) ↔ $9 (register 9) → 01001

constant : 14 → 1110

op (6 bits)	rs (5 bits)	rt (5 bits)	constant (16 bits)
000100	01000	01001	0000 0000 0000 1110
beq	$8	$9	14

MIPS J-Type Instruction



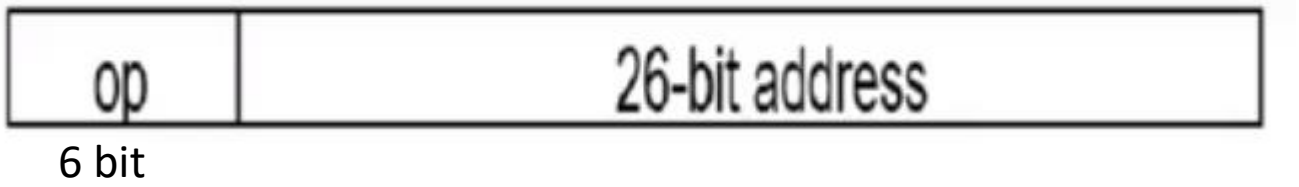
1. **j loop** : jump to loop; to the line of code that has loop address,
2. **jal loop**: jump and link to loop address,

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
		op	address					
j	J	2	2500					j 10000
jal	J	3	2500					jal 10000

- Difference between j and jal: j (jump) places a certain address in the program counter and continues execution from there, however, jal (jump and link) does the same but it saves the return address in ra so that you can continue execution after your subroutine finishes.

```
int main() {
    dosomething();
    //code here
}


void dosomething() {
//.....code here
}
```

MIPS J-Type Instruction



1. **Example** j 14 → jump to loop address 14; op code 2

op code 2 → 000010

address → 14 (decimal) → 1110

op code	26 bit address
000010	00 0000 0000 0000 0000 0000 1110

2. **Example** jal 14 → jump and link to loop address 14; op code 3

op code 3 → 000011

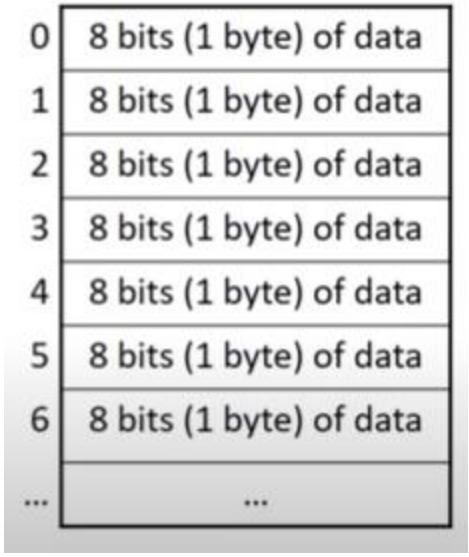
address → 14 (decimal) → 1110

op code	26 bit address
000011	00 0000 0000 0000 0000 0000 1110

MAIN MEMORY

- Code and data storage: we use memory to store the code and data for a program which the processor fetches and loads and store to get data in/out of registers and to move instructions into the processor for execution.
- Memory is **Byte-addressed:** meaning that the indexes in memory are all number of eight bits (one byte).

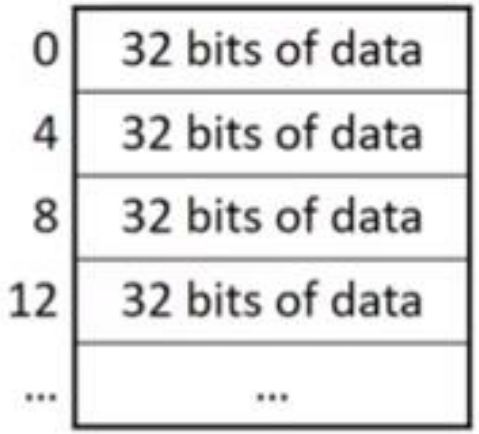
- MIPS Memory is
- 1. Word aligned
- 2. Big Endian



MIPS Memory: Word Aligned "Words of Memory"

- Word is size of instruction
- Typically 4 bytes (8 bytes in 64 processor)

Word alignment means that every memory address starts at a multiple of the word . So in MIPS every memory access has to start at a multiple of four.



MIPS: Byte Addressing

- Computers are divided into those that use the address of the left most or “big end” byte as the word address versus those that use the rightmost or “little end” byte. This Depends on whether bits or bytes or other components are ordered from the big end (most significant bit) or the little end (least significant bit).
- MIPS is in the *big-endian camp. Since the order matters only if you access the identical data* both as a word and as four bytes, few need to be aware of the endianess.
- Byte addressing also affects the array index. To get the proper byte address in the code above, *the off set to be added to the base register $s3 must be 4 8, or 32, so* that the load address will select A[8] and not A[8/4].
- In MIPS, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**

Big Endian

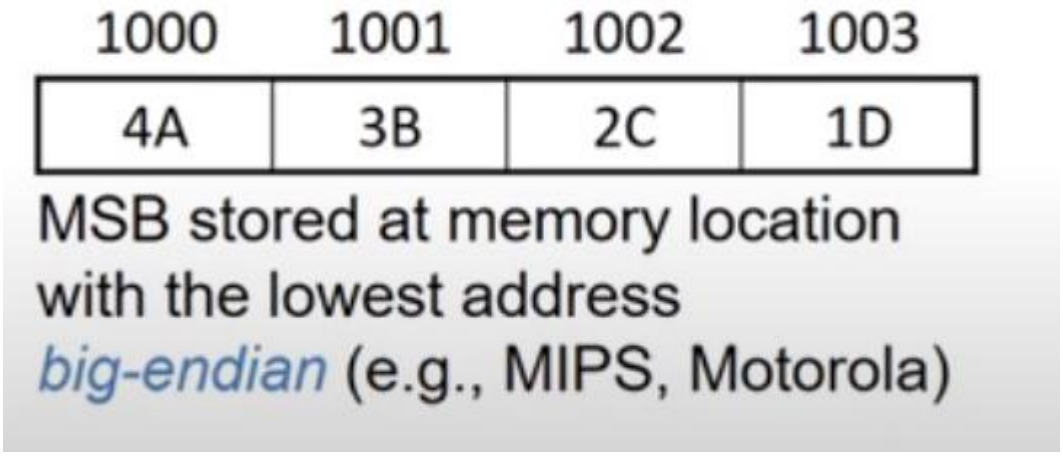
- Most significant byte at smallest address in memory.
- “The Big End comes first”: means that when you look at the order of the bytes in word the most significant byte starts at the smallest address
- Alternative: Little Endian – Big-Endian

Endianness

Example: storing the 32-bit **hexadecimal integer** value

4A3B2C1D in memory at address 1000:

- In a big Endian architecture the first part of the word will be stored first i.e. at the smaller address



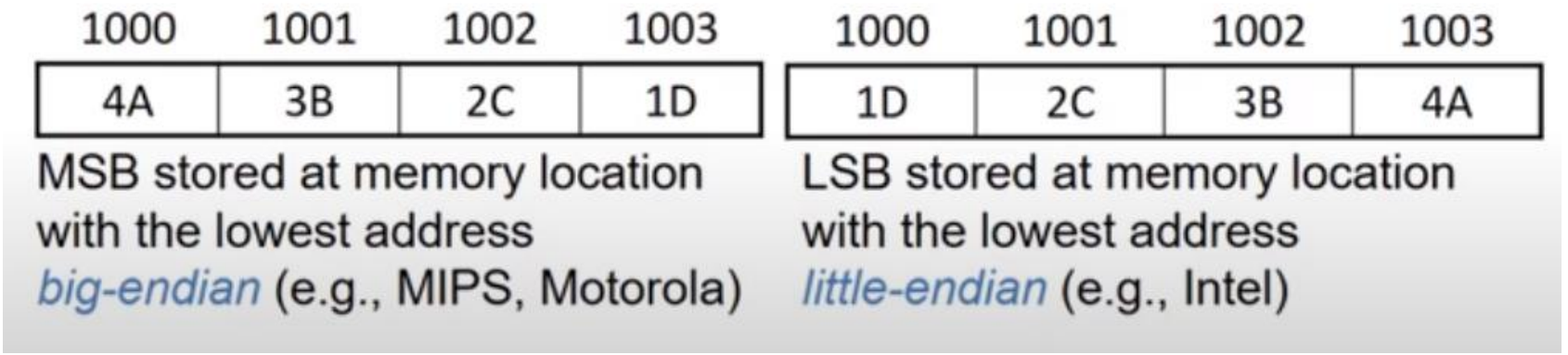
Note: ever two characters in the hexadecimal word is one byte because each character of hexadecimal is four bits (half byte or nibble)

Endianness

Example: storing the 32-bit **hexadecimal integer** value

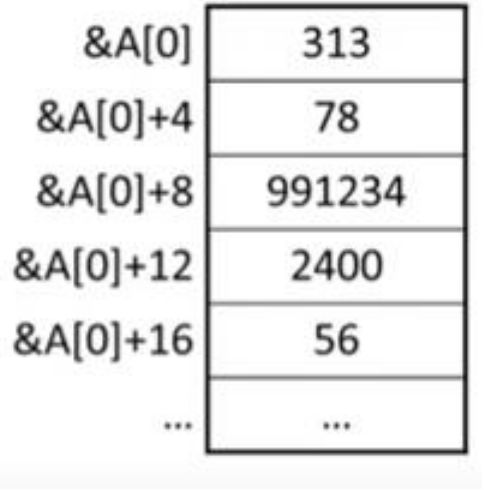
4A3B2C1D in memory at address 1000:

- In a little Endian architecture the first part of the word will be stored last

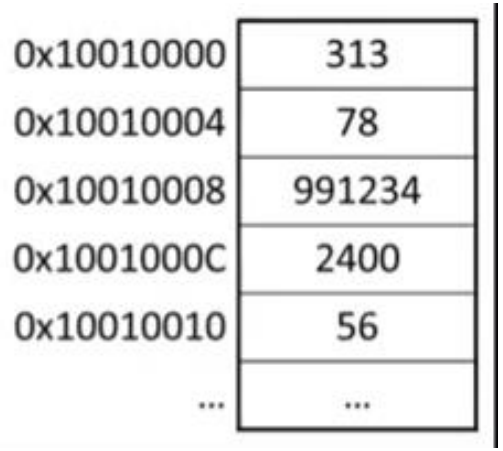


Arrays in Memory

- Arrays are put in memory in the order in which the array elements appear.



- Suppose that the starting address of A (hexadecimal): 0x10010000



Arrays in Memory

- The equivalent value of 313 (dec) = 00000139 hex

Index	0
Address (hex)	10010000
Content (dec)	313
Content (hex, BE)	00000139
Content (hex, LE)	39010000

0x10010000	313
0x10010004	78
0x10010008	991234
0x1001000C	2400
0x10010010	56
...	...

313 (dec) = 00000139 (hex)

4 bytes = one word

Arrays in Memory

```
int A[5] = {313, 78, 991234, 2400, 56};
```

Index	0	1	2
Address (hex)	10010000	10010004	10010008
Content (dec)	313	78	991234
Content (hex, BE)	00000139	0000004E	000F2002
Content (hex, LE)	39010000	4E000000	02200F00

0x10010000	313
0x10010004	78
0x10010008	991234
0x1001000C	2400
0x10010010	56
...	...

Signed/Unsigned numbers

- Computer programs calculate both positive and negative numbers.

In MIPS, signed numbers range:

- Positive numbers, from 0 to $2{,}147{,}483{,}647_{10}$ (2^{31} -1),
- Negative numbers range from (-2^{31}) to -1.

- How to distinguish the positive from the negative numbers?

solution

- Use *sign and magnitude:* add a separate sign bit (single bit); (0 = positive, 1 = negative) and the remaining bits are the magnitude.

Example:

$+25_{10} = 00011001_2$ $\qquad$ $-25_{10} = 10011001_2$

Signed/Unsigned numbers

Shortcomings (limitations):

1) It's not obvious where to put the sign bit. To the right? To the left ? Early computers tried both.
2) adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be.
3) A separate sign bit means that sign and magnitude has both a positive and a negative zero (-0, +0), which can lead to problems for inattentive programmers.

- sign and magnitude had proble $+0 = 00000000_2$, $-0 = 10000000_2$. er

As a result of these shortcomings, sign and magnitude representation was soon abandoned (تم التخلي عنها).

Load Byte Signed/Unsigned

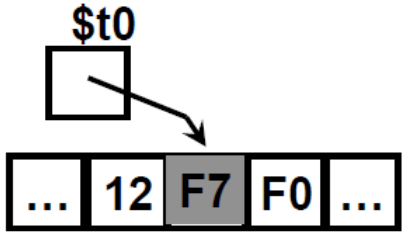
In MIPS:
Signed binary numbers representation is by:

- Leading 0s mean positive,
- leading 1s mean negative.

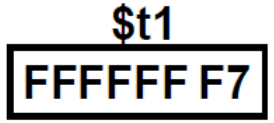
This convention is called ***Two's complement***.

$$
\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}\\
&\ldots \qquad\qquad \ldots\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}\\
&\ldots \qquad\qquad \ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}
\end{aligned}
$$

Note: in a MIPS word
least significant bit
The rightmost bit.
most significant bit
The left most bit.



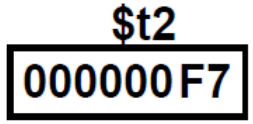
lb $t1, 0($t0)



Sign-extended

Example →

lbu $t2, 0($t0)



Zero-extended

every computer today uses two's complement for signed no.s

MIPS Instructs

Addressing modes
Loading larger values
Logical instructions
Examples : compiling C programing segments to MIPS program segments

MIPS Operands and Instructions

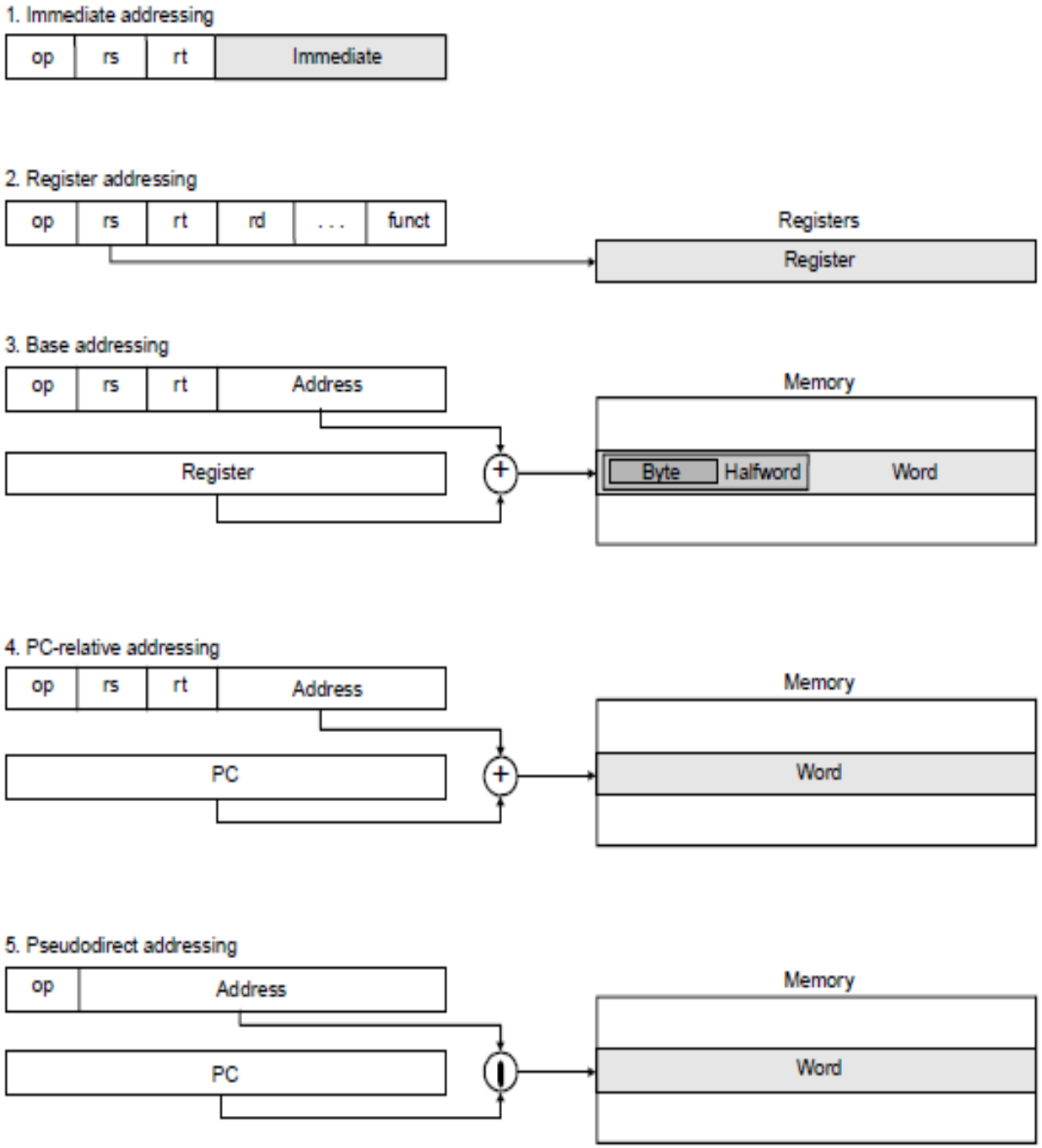
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{32} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Addressing Mode



Addressing Modes

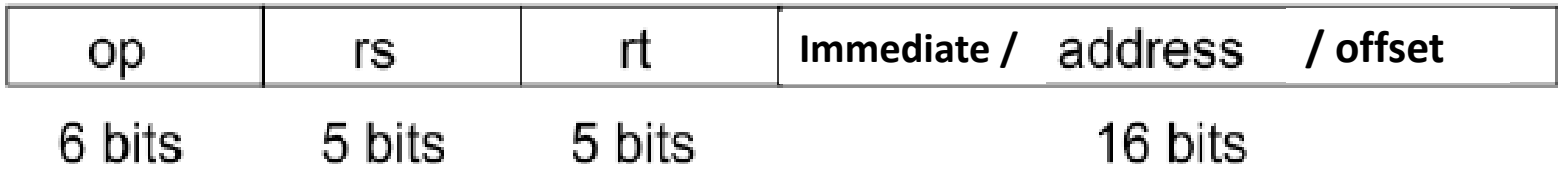
What are the different ways to access an operand?

- **Immediate addressing:** One of the operand values is immediate / a constant (obtained at run time).
  - Immediate arithmetic
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

Example: addi $s0, $zero, 7 7 is an immediate value

addi $s1, $zero, 7 *means* $s1 ← 0 + 7

I-Format



Addressing Modes

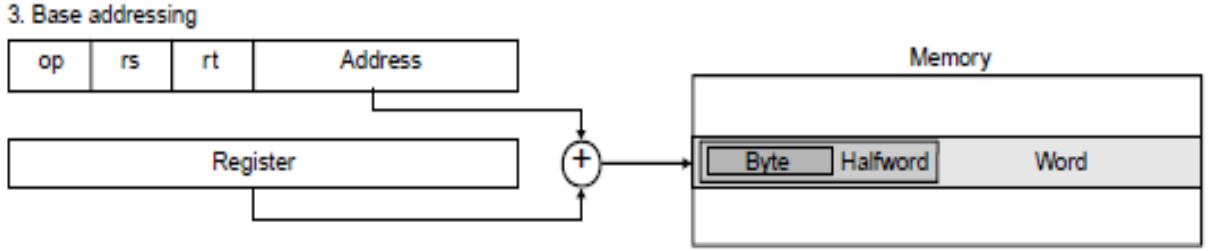
- **Register addressing:** All the operands are registers

  add $s1, $s2, $s3 means $s1 ← $s2 + $s3

- **Base addressing:** One of the operands is memory and the other operand is a register. (Data transfer instructions come under base addressing).

Examples: lw $s1, 32($s2) → $s1 = Mem [$s2 + 32]

sw $s0, 8($s4) → Mem [$s4 + 8] = $s0



lw $s1, 32($s2) means $s1 ← M[$s2 + 32]

- **Direct addressing** $s1 ← M[32] *We are going to access the data from memory directly*
- **Indirect addressing** $s1 ← M[$s2] *Helps implement pointers.*

Addressing Modes (Cont.)

- **PC-relative addressing:** Implements position-independent codes. A small offset is adequate for short loops.

Effective address **=PC + 4+ (4 * offset)** determines the target address

Example1 : bne $s0, $s1, Label # **Label represent the offset**

Let PC=2000; Label= 2500;

- ***The effective address*** = 2000+4+(4*2500) #PC relative address

bne $s0, $s1, 2500 if ($s0 != $s1) go to PC+4+(4*2500)

Example2: **beq $s0, $s1, 20**

if ($s0 = = $s1) go to PC + 4 + 80

Addressing Modes (Cont.)

- **Pseudo-direct addressing:** Used in the J-format.
- The target address is (4*offest)

E.g1. j 2500 go to 10000 *(effective address is 4*2500)*

E.g2. jr $ra go to $ra *(effective address is given inside $ra)*

E.g.3 jal 2500 $ra = PC + 4 then go to 10000

we have to jump to the address 10000 but we have to save the return address (PC+4) inside $ra, after processing the task in , the execution will come back to the ra

Loading Larger Values

The immediate field is limited to 16 bits in I-Format (-32,768 to 32,767)

How do we load larger values?

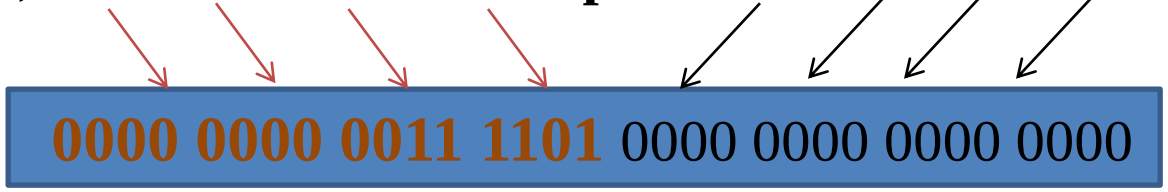
Use two instructions to combine two 16 bit immediates

- Load Upper Immediate (lui) : Loads upper 16 bits
- Or immediate (ori): Loads lower 16 bits

Example: load the 32-bit constant into register $s0

0000 0000 0011 1101 0000 0000 0000 0101

1) lui $s0, **0000 0000 0011 1101 put zeros in the lower bits**



2) ori $s0, **0000 0000 0000 0101**

Loading Larger Values

1) lui $s0, **0000 0000 0011 1101**

lui $s0, 61 [61 decimal = **0000 0000 0011 1101 (binary)]**

2) ori $s0, **0000 0000 0000 0101**

ori $s0, $s0, 5 # 5 decimal = 0000 0000 0000 0101

The final value in register $s0 is the desired value:

0000 0000 0011 1101 0000 0000 0000 0101

Question: what is the MIPS code to load the 32-bit constant

0000 0000 0011 1101 0000 0000 0000 0101 into register $s0?

1. **lui $s0, 61** (op code 15 decimal, no source, rt(destination) $s0: $16, immediate 61 decimal)

ntains

0000 0000 0000

op (6 bits)	rs (5 bits)	rt (5 bits)	Immediate (16 bits)
001111	00000	10000	**0000 0000 0011 1101**

2. ori $s0, $s0, 5 (op code 13(dec), rs: $s0, rd: $s0, immediate: 5

ains the

op (6 bits)	rs (5 bits)	rt (5 bits)	Immediate (16)
001101	10000	10000	**0000 0000 0000 0101**

Logic Instructions

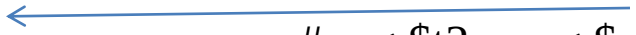
Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

shifts. They move all the bits in a word to the left or right, filling the emptied bits with 0s.

shift left logical **(sll):** For example, if register $s0 contains

0000 0000 0000 0000 0000 0000 0000 $1001_2 = 9_{10}$

0000 0000 0000 0000 0000 0000 1001 0000



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

If the instruction to shift left by 4 was executed, the new value would be:

0000 0000 0000 0000 0000 0000 1001 $0000_2 = 144_{10}$

If you want to multiply by 2 then you need to shift left by 1, sll $t2,$s0,1

If you want to multiply by 4 then you need to shift left by 2, sll $t2,$s0,2

Logic instructions: And, or, nor, andi

If register $t2 contains 0000 0000 0000 0000 0000 1101 1100 0000_2
register $t1 contains 0000 0000 0000 0000 0011 1100 0000 0000_2

1. or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2

$t2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	0
$t1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
$t0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	0	0	0	0	0

Register $t0= 0000 0000 0000 0000 0011 1101 1100 0000

2. and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2

$t2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	0
$t1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
$t0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

Register $t0= 0000 0000 0000 0000 0000 1100 0000 0000

Logic instructions: And, or, nor

If register $t2 contains 0000 0000 0000 0000 0000 1101 1100 0000_2
register $t1 contains 0000 0000 0000 0000 0011 1100 0000 0000_2

3. nor $t0,$t1,$t2 # reg $t0 = ¬ (reg $t1 | reg $t2)

¬ (0000 0000 0000 0000 0011 1101 1100 0000)

register $t0: 1111 1111 1111 1111 1100 1110 0011 1111

4. andi $t0,$t1, 7 # reg $t0 = reg $t1 & 7

7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
$t1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
$t0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instructions for Making Decisions

- beq register1, register2, L1 # branch if register1= register2

  beq→ *branch if equal*.

- bne register1, register2, L1

  If register1≠ register2 goto L1 # go to the statement labeled L1

  Bne→ *branch if not equal*.

Example: Compile the following instruction

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

f = $s0; **g** = $s1; **h** = $s2; **i**= $s3; **j**= $s4

In MIPS

```
bne $s3,$s4,Else          # go to Else if i ≠ j
add $s0,$s1,$s2           # f = g + h (skipped if i ≠ j)
j Exit                    # go to Exit
Else: sub $s0,$s1,$s2     # f = g – h (skipped if i = j)
Exit:
```

Pseudo-instructions

Pseudo-Instruction are simple assembly language instructions that do not have a direct machine language equivalent. During assembly, the assembler translates each pseudo instruction into one or more machine language instructions.

- Pseudo-instructions give MIPS a richer set of assembly language instructions

Example:

```
move $t0, $t1     # $t0 ← $t1
```

The assembler will translate it to

```
add $t0, $zer0, $t1
```