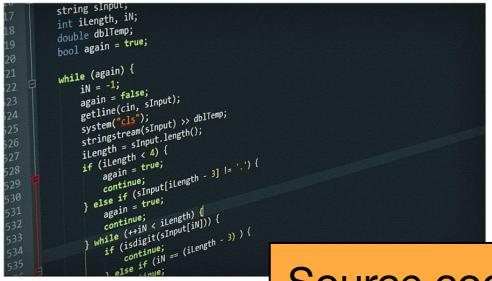
Computer Architecture

Instructions: the Language of Computer

the MIPS instruction set, Part 1

Program execution from the view of entry level programmers

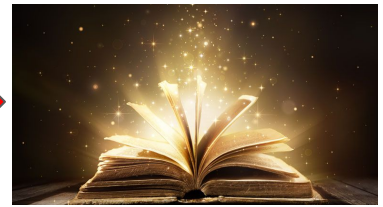


Source code

Compilation

```
add t0, t1, t2
lw t3, 16(t0)
slt t0, t1, 0x6eb21
```

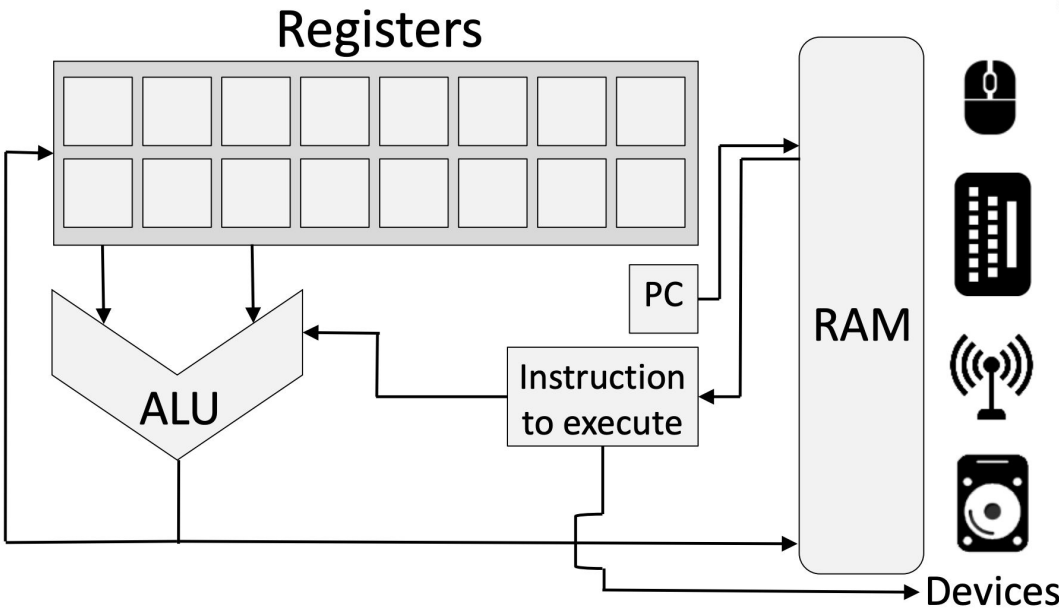
Machine instructions



A hardware magic occurs

Answers and outputs

Program execution from the view of mediocre programmers



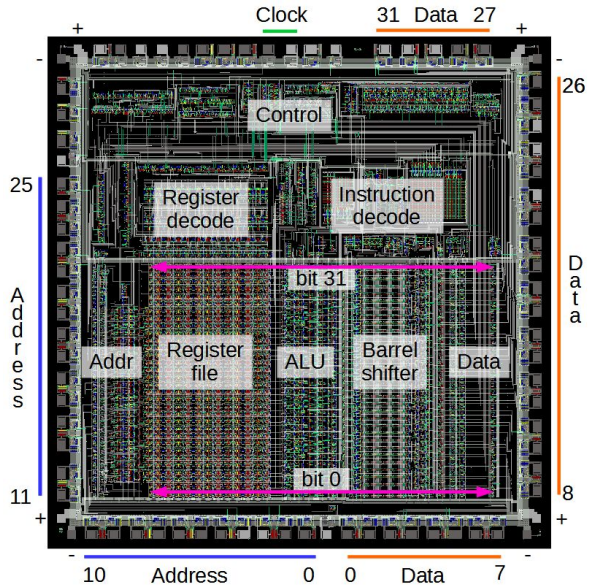
- Program instructions live in RAM
- PC register points to the memory address of the instruction to fetch and execute next
- Arithmetic logic unit (ALU) performs operations on registers, writes new values to registers or memory, generates outputs which determine whether to branches should be taken
- Some instructions cause devices to perform actions

A few things still need to be explored in details

What happens when an instruction is executed by the circuits?

- How does an instruction control the circuits?
- How do the circuits recognize an instruction?
- How do the circuits recognize numbers?
- How does the ALU work?
- ……

We will start the journey with the MIPS, an instruction architecture sets (ISAs).



Instruction Set Architectures (ISAs)

- ISA defines the interface which hardware presents to software.
  - A compiler translates high-level source code (e.g., C, C++) to the ISA for a target processor
  - The processor directly executes ISA instructions
- Example ISAs:
  - MIPS (mostly the focus of this course)
  - ARM (popular on mobile devices)
  - x86 (popular on desktops and laptops)
- Different processors are built based on different ISAs, but with many aspects in common.
  - Even for a single ISA, there can be different hardware implementations. (e.g., Intel and AMD both build processors of x86 architecture, but they use different underlying hardware.)

Agenda

- MIPS instruction set in assembly language
  - Opcode and operand
- Representing data in binaries
  - Signed and unsigned numbers
- Representing MIPS instructions in binaries (machine language)
  - R-format
  - I-format
  - J-format
- Stored program computers

The MIPS Instruction Set in Assembly Language

MIPS is typical of many modern ISAs. Each assembly language instruction of MIPS always contains two parts:

- The operation to perform, and
- The operands on which to operate.

In general, operations can be categorized into arithmetic, logical, data transfer, and branching, while the operands are defined as three types: register, memory, and constant.

The design principles of the MIPS architecture

Principle 1: **simplicity favors regularity**

- Every instruction of MIPS is of length 4 bytes= 32 bits.

Principle 2: smaller is faster

- Only 32 registers in the MIPS architecture.

Principle 3: make the common case fast

- MIPS provides immediate operands for small constants, which avoids load instructions

Principle 4: good design demands good compromises

- The instructions are separated into three different formats: R, I, and J.

Arithmetic Operations

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants

- Add and subtract, three operands
  - Two sources (b, c) and one destination (a)

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

- All arithmetic operations have this form (recall the **simplicity favors regularity)**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- **C code:**

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

- **Compiled MIPS code:**

```
add t0, g, h     # temp t0 = g + h
add t1, i, j     # temp t1 = i + j
sub f, t0, t1    # f = t0 - t1
```

Operands in assembly instructions

In assembly, the operands are not variable names but essentially physical locations where the instructions should retrieve the binary data. These physical locations can be *registers*, *memory locations*, or *in the instructions* itself (some instructions contain constants).

Registers in the MIPS architecture

- There are 32 registers, and each has the capacity to store 32 bits.
  - Recall the **smaller is faster**: registers are much faster than main memory which has millions of locations, because it takes electronic signals longer to travel farther.
- So, in MIPS, a “word” means a sequence of 32 bits.
  - The term “word” generally refers to the standard size of data that a CPU's register can handle in a single operation; for MIPS, it is 32-bit (i.e., 4 bytes).

Registers operands in MIPS

By convention, each register has it own name, index, and status or function.

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

Note: `Register 1`, named `$at`, is reversed for the assembler, and `register 26-27`, named `$k0-$k1`, are reversed for the operating system.

Compiling a C Assignment Using Registers

- **C code:**

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

- **Compiled MIPS code:** (Assume `f, g, h, i, j` are assigned to `$s0,…,$s4`.)

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

add $t1, $s3, $s4

sub $s0, $t0, $t1
```

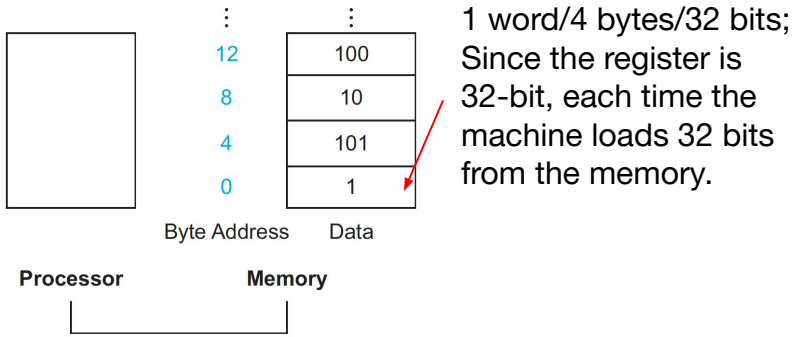
Memory Operands

There are complex data structures, e.g., arrays and structures in C, which can contain many data elements than there are registers in a computer.

- Main memory used for representing and accessing composite data
  - Processor can only keep a small amount of data in registers but complex data structures in memory.
  - Memory has many data locations but accessing it takes a long time.
- To apply arithmetic operations in such cases, we need data transfer instructions. With them, we can
  - load values from memory into registers, and
  - store result from register to memory

Memory

Memory is a large, single-dimensional array; memory addresses are like indices to that array, starting at 0 (as the following figure shows).



1 word/4 bytes/32 bits; Since the register is 32-bit, each time the machine loads 32 bits from the memory.

- Memory is *byte addressable*: each address identifies an 8-bit byte.
- `lw`: read a word from memory into a register;
  - `lw $s1, 20($s2)` #load a word (4 bytes) starting from the address (`$s2 + 20`), `$s2` stores the base address, 20 is the offset (in bytes).
- `sw`: write a word from a register to memory
- Words are aligned in memory
  - A word is 4 bytes (i.e., 32 bits); so, the address of a word is the address of first one of the 4 bytes and **must** be a multiple of 4.
  - `lw $s1, 7($s2)` is an illegal instruction in MIPS because the offset is not aligned (i.e., not a multiple of 4)

Memory Operand Example 1

Assume `A` is an array of 100 words,

- **C code:** `g = h + A[8]`
  - `g` in `$s1`, `h` in `$s2`, base address of `A` is `$s3`
  - Base address: the address of the first element, `A[0]`.
- **Compiled MIPS code**:

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

  - Index 8 means 8 words, which requires offset of 32 bytes from `A[0]`.
  - Offset should be counted in bytes since MIPS is byte-addressable

Memory Operand Example 2

- C code: `A[12]= h + A[8]`
  - `h` in `$s2`, base address of `A` is `$s3`
  - Base address: the address of the first element, `A[0]`.
- Compiled MIPS code:

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

  - Index 8 means 8 words, which requires offset of 32 bytes from `A[0]`.
  - Index 12 requires offset of 48 bytes.

Big Endian vs. Little Endian

- Big-Endian: Most significant byte (MSB) is stored at the lowest memory address.
- Little-Endian: Least significant byte (LSB) is stored at the lowest memory address.

- MIPS is Big-Endian
- The difference between big-endian and little-endian matters when dealing with data exchange between different systems, file formats, or low-level programming like network communications and binary file parsing.

Registers vs. Memory

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

About register optimization

- **Motivation:**
  - Many programs have more variables than computers have registers. So, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory.
  - Data is more useful when in a register: A MIPS arithmetic instruction can read two registers and operate on them while a data transfer instruction only read or write one operand.
- **Goals of the optimization**:
  - an instruction set architecture must have a sufficient number of registers, and
  - compilers mush use registers efficiently
- The process of putting less commonly used variables (or those needed later) into memory is called **spilling registers**.

Immediate Operands (constant)

- Constant data specified in an instruction

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

- No subtract immediate instruction ⇒ just use a negative constant

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

- Design Principle 3: **Make the common case fast**
  - Small constants are common (e.g., the `i++` in a C for loop)
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0. (It cannot be overwritten.)
- Useful for common operations
  - E.g., move between registers: `add $t2, $s1, $zero`

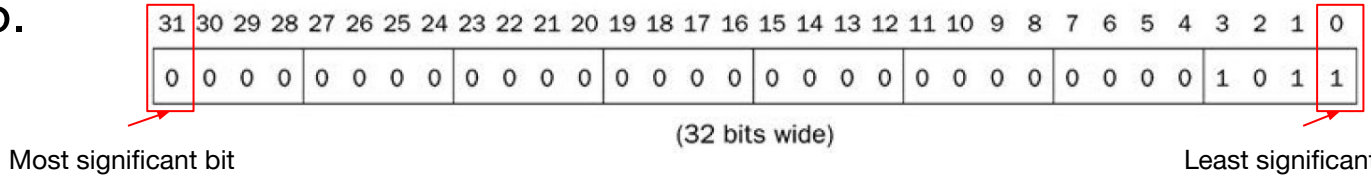
Unsigned Binary Integers

- A single digit (i.e., a bit) of a binary number is the “atom” of computing.
- Given an n-bit binary number, the corresponding decimal number is

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

Range: $0 \text{ to } 2^n - 1$

For a 32-bit binary number, it can represent any decimal within 0 to +4,294,967,295.



Convert unsigned binary to decimal

Example:

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

  $= 0 + \ldots + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

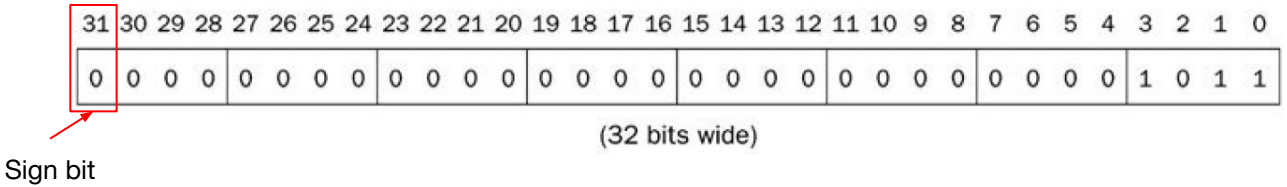
2s-Complement Signed Integers

- The most significant bit (i.e., the bit on the rightmost) is set to be the sign bit.
  - If 0 the number is non-negative (from 0 to $2^{n-1}-1$)
  - If 1 the number is negative
- The binary of all 0s is defined to be 0
  - Positive values are obtained by starting with a string of 0s of a preset length and counting in binary until the pattern consisting of a single 0 followed by 1s
  - Negative values are obtained by starting with a string of 1s of the preset length and counting backward in binary until the pattern consisting of a single 1 followed by 0s.
- The binary of 1000…000 is defined to be the most negative number.

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

2s-Complement Signed Integers



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

2s-Complement Signed Integers

- Given an n-bit number in two's complement form, we have

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

Range: -2^{n-1} to $2^{n-1} - 1$ (Note: it is not symmetric.)

- Example: a 32-bit binary
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

$$= -1x2^{31} + 1x2^{30} + \ldots + 1x2^2 + 0x2^1 + 0x2^0$$

$$= -2,147,483,648 + 2,2147,483,644 = -4_{10}$$

Signed Negation

- A quick way to negate a two's complement binary number: invert every 0 to 1 and every 1 to 0, then add one to the result.

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2

$+2 = 0000\ 0000\ \ldots\ 0010_2$

$-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$

$= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

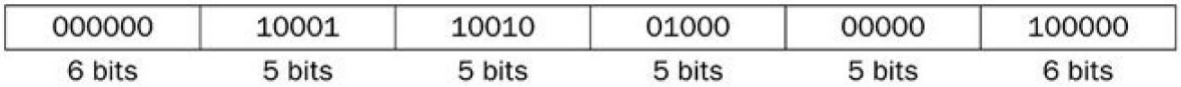
- Representing a number using more bits but preserve the numeric value
- In MIPS instruction set (sometimes the data to process may not be 32-bit long)
  - `addi`: extend immediate value
  - `lb, lh`: extend loaded byte/halfword
  - `beq, bne`: extend the displacement
- Replicate the sign bit to the left
  - E.g., 8-bit to 16-bit
    - +2: 0000 0010 ⇒ 0000 0000 0000 0010
    - -2: 1111 1111 ⇒ 1111 1111 1111 1110

Representing Instructions in Binary

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

Representing Instruction in Binaries

- Instruction format: the layout of the instruction
- Instruction formats of MIPS:
  - R-format (for register)
  - I-format (for immediate)
  - J-format (unconditional jump)
- An instruction is composed of fields of binary numbers, also called machine code.



000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- We often use hexadecimal to represent the binaries in machine code

Hexadecimal

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

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

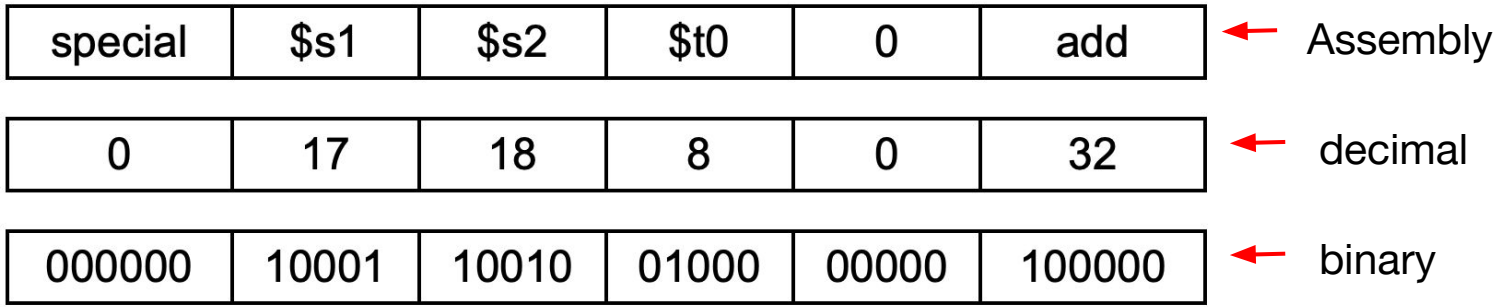
FIGURE 2.4 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

MIPS R-format instructions

- Instruction fields:
  - `op`: operation code (opcode)
  - `rs`: first source register number
  - `rt`: second source register number
  - `rd`: destination register number
  - `shamt`: shift amount (using in logical operations)
  - `funct`: function code (extends opcode)

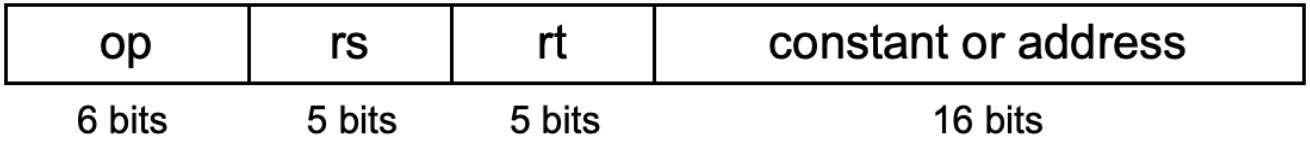
R-format Example

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



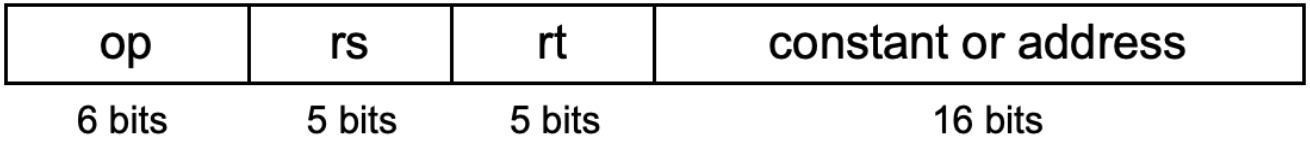
$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 02324020_{16}$

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - `rs`: the first source register
  - `rt`: destination or source register number
  - `constant`: -2^{15} to $+2^{15}-1$
  - `address`: offset added to base address in `rs`
- 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes (or, 8192 words) of the address in the register `rs`.

I-format Example

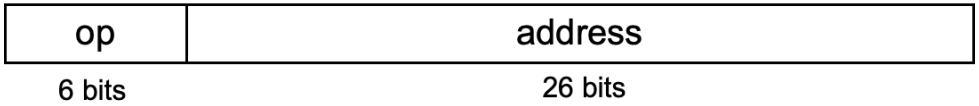


- Assume the base address of `A` is in `$s3`, we want to load `A[8]` into `$t0`

```
lw $t0, 32($s3)     #Temporary reg $t0 gets A[8]
```

lw	$s3	$t0	32
35	19	8	32
100011	10011	01000	0000000000100000

J-format Instruction



- Jump (`j` and `jal`) targets could be anywhere in text segment.
  - `address`: the target address

- *Design Principle 4*: **Good design demands good compromise**
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

More Examples

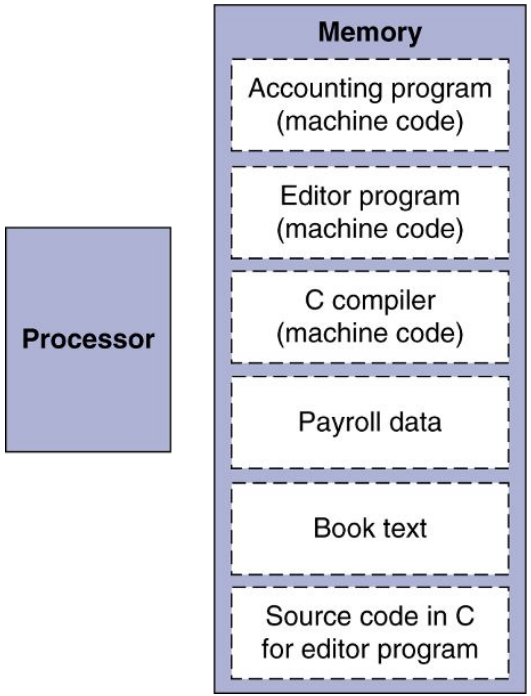
MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

Stored Program Computers

The BIG Picture



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

Exercise 1

What MIPS instruction does this represent? (choose from the four options)

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. `sub $t0, $t1, $t2`
2. `add $t2, $t0, $t1`
3. `sub $t2, $t1, $t0`
4. `sub $t2, $t0, $t1`