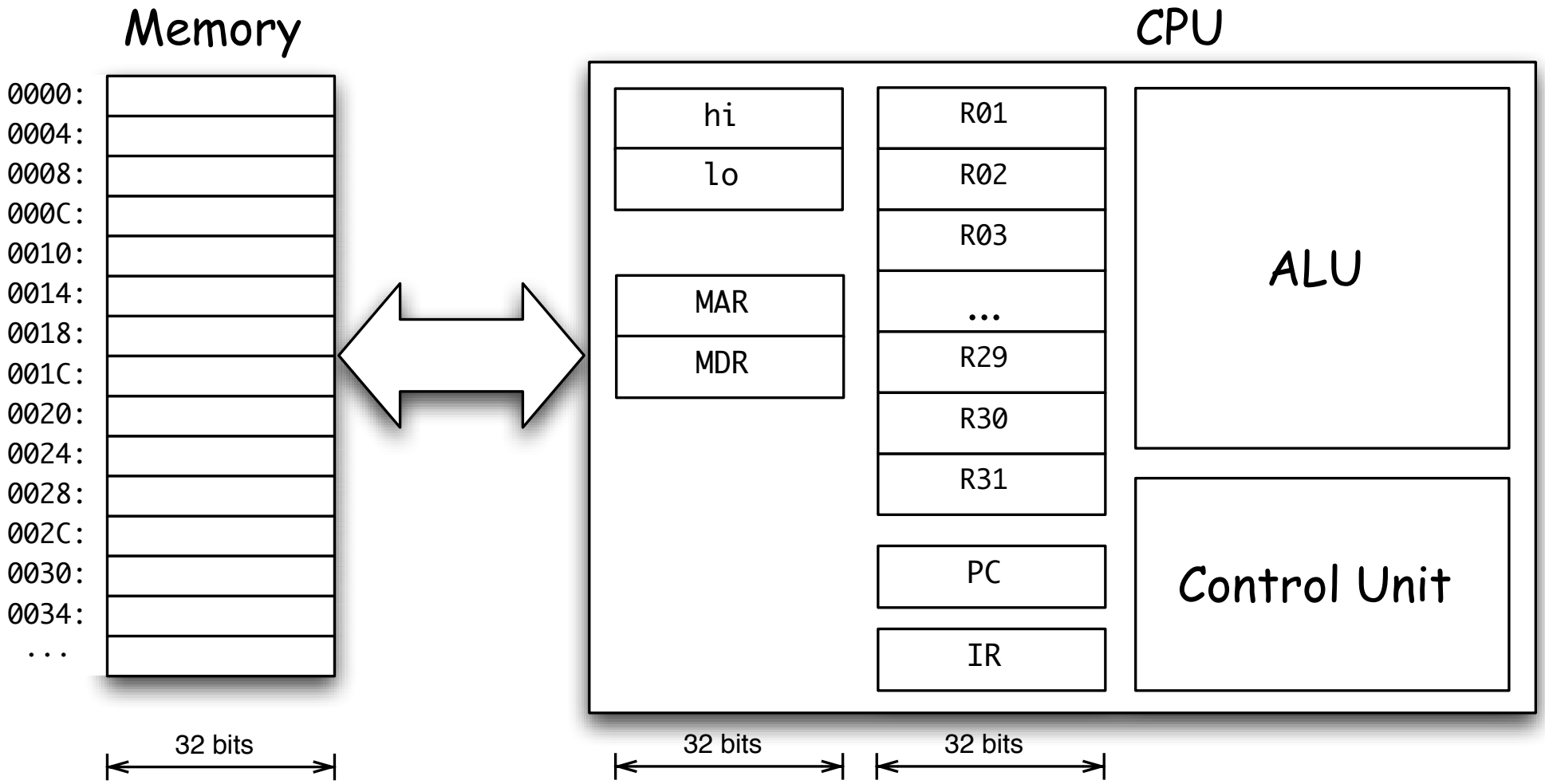
MIPS Architecture



Expect to use Linux in this course!

What does a data look like in memory?

What does a machine instruction look like in memory?

What does a data look like in memory?

What does a machine instruction look like in memory?

- Data is stored as binary sequences (1's and 0's).
- Machine instructions are also binary sequences.
- Only certain sequences of 1's and 0's are valid machine instructions.
- Machine languages are processor specific.

If both programs and data are stored in memory, how do we tell them apart?

What does a data look like in memory?

What does a machine instruction look like in memory?

- Data is stored as binary sequences (1's and 0's).
- Machine instructions are also binary sequences.
- Only certain sequences of 1's and 0's are valid machine instructions.
- Machine languages are processor specific.

If both programs and data are stored in memory, how do we tell them apart?

Recall from last class, by simply looking at the bits, we can't!

Basic Architecture

- Computer programs operate on data but are data themselves.
- Both, a program and the data it operates on are stored in the same memory.
- Fundamentals: Random-access memory (RAM), CPU, fetch-execute cycle, stored program.

Recall, from CS 135 history, who this architecture is named after?

Basic Architecture

- Computer programs operate on data but are data themselves.
- Both, a program and the data it operates on are stored in the same memory.
- Fundamentals: Random-access memory (RAM), CPU, fetch-execute cycle, stored program.

Recall, from CS 135 history, who this architecture is named after?

Single-memory, stored program architecture is known as von Neumann architecture after John von Neumann.

Also invented merge sort in 1945.

CS 241 Simplified MIPS

- Central Processing Unit (CPU)
- Control Unit
  - Decodes instructions
  - Interacts with other components of CPU to carry them out
- Arithmetic Logic Unit (ALU)
- Memory - lots of different kinds

Memory

- CPU registers (fast, expensive)
- L1, L2, L3 cache
- RAM (main memory)
- Solid state drive (SSD), harddrive
- Networks, The Cloud (slow, cheap)

We will focus on registers and RAM.

Registers

A CPU has a very limited number of very fast, but very small memory storage locations called *registers*.

Our MIPS architecture has 32 registers, 32 bits in size; one *word*.

Registers

A CPU has a very limited number of very fast, but very small memory storage locations called *registers*.

Our MIPS architecture has 32 registers, 32 bits in size; one *word*.

- CPU can only operate on data stored in registers.
- If data is not currently in a register, you must load it from RAM.
- Registers are labelled $0, $1, $2, . . ., $31
- $0 is always 0
- $31 is special, $30 is also special, $29 is sort of special

How many bits are needed to encode a register number in binary?

32 registers. $32 = 2^5 \Rightarrow 5$ bits needed to encode a register number.

First MIPS Instruction - Add

An example of a MIPS instruction is:

Add contents of registers *s* and *t*, store the result in register *d*.

MIPS Reference Sheet: `add $d, $s, $t`

Binary: `0000 00ss ssst tttt dddd d000 0010 0000`

Example: `add $27, $7, $8`

First MIPS Instruction - Add

An example of a MIPS instruction is:

Add contents of registers *s* and *t*, store the result in register *d*.

MIPS Reference Sheet: `add $d, $s, $t`

Binary: `0000 00ss ssst tttt dddd d000 0010 0000`

Example: `add $27, $7, $8`

- 7 in 5-bit binary is: `00111`
- 8 in 5-bit binary is: `01000`
- 27 in 5-bit binary is: `11011`

Final: `0000 0000 1110 1000 1101 1000 0010 0000`

Conventions

For readability, group binary strings into blocks of 4 bits.

For convenience, use hexadecimal (base 16) instead of binary.

- Digits: 0, 1, . . . , 9, a, b, c, d, e, f
- 1 hexadecimal digit corresponds to 4 binary bits (and vice-versa)
- Denote hexadecimal numbers by starting them with `0x`

Binary: `0000 0000 1110 1000 1101 1000 0010 0000`

Hexadecimal:

Conventions

For readability, group binary strings into blocks of 4 bits.

For convenience, use hexadecimal (base 16) instead of binary.

- Digits: 0, 1, . . . , 9, a, b, c, d, e, f
- 1 hexadecimal digit corresponds to 4 binary bits (and vice-versa)
- Denote hexadecimal numbers by starting them with `0x`

Binary: `0000 0000 1110 1000 1101 1000 0010 0000`

Hexadecimal: `0x00e8d820`

Another common number system is called Octal (base 8).

Exercise: convert `0xcab` into Octal.

RAM

- Big array of **n** bytes (**n** is large), away from the CPU
- Each cell has an address 0, 1, 2, . . ., **n**-1
- Each 4-byte block $4k$, . . ., $4k+3$ (for k=0,1,. . .) is a word
- Words have addresses `0x0, 0x4, 0x8, 0xc, 0x10, 0x14, 0x18, 0x1c, 0x20, ...`
- Known as *word aligned*; i.e. divisible by 4

Data in RAM must be loaded into registers before the CPU can use it!

RAM access is very slow compared with register access.

What does your CPU while it waits?

In a simple scenario, the CPU does nothing and simply waits for the needed data to continue. You may assume this behaviour in CS 241.

In some architectures, instead of simply doing nothing, your CPU may switch to work on another thread.

Communicating with RAM

Two operations: load and store

Load transfers a word from a source address in RAM into a target register.

MIPS Reference Sheet: `lw $t, i($s)` and `sw $t, i($s)`

Binary: `1000 11ss ssst tttt iiii iiii iiii iiii`

- `i($s)` = base address `$s` + offset `i` (RAM is an array)
- `i` is an integer

How do we encode an integer value into binary?

Communicating with RAM

Two operations: load and store

Load transfers a word from a source address in RAM into a target register.

MIPS Reference Sheet: `lw $t, i($s)` and `sw $t, i($s)`

Binary: `1000 11ss ssst tttt iiii iiii iiii iiii`

- `i($s)` = base address `$s` + offset `i` (RAM is an array)
- `i` is an integer (in bytes)

How do we encode an integer value into binary?

Two's complement

In the load instruction, `i` is an *immediate* value.

An *immediate* value is one that is encoded directly into the MIPS instruction (code).

When encoding MIPS instructions:

- Register numbers are encoded as unsigned binary.
- Immediate values are encoded as two's complement binary (often using 16 bits).

MIPS Reference Sheet: `lw $t, i($s)`

Binary: `1000 11ss ssst tttt iiii iiii iiii iiii`

Example: Given `$1` stores the base address of an array and `$2` stores the length, load the value at index 7 into `$3`.

- Note: There are 16 bits reserved for integer `i`.
- Size of an element is 4 bytes

MIPS Reference Sheet: `lw $t, i($s)`

Binary: `1000 11ss ssst tttt iiii iiii iiii iiii`

Example: Given `$1` stores the base address of an array and `$2` stores the length, load the value at index 7 into `$3`.

- Note: There are 16 bits reserved for integer `i`.
- Size of an element is 4 bytes $\Rightarrow$ `i` is $7 \times 4 = 28$ bytes
- 28 in 16-bit binary is: `0000 0000 0001 1100`

Assembly: `lw $3, 28($1)`

Final: `1000 1100 0010 0011 0000 0000 0001 1100`

The load and store operations use two special registers that are part of the CPU.

- The address of data to load from RAM is stored in Memory Address Register (MAR).
- The address then goes on the bus and is delivered to RAM.
- Data from that location is then returned on the bus and stored in Memory Data Register (MDR).
- Contents of MDR are then moved to target register.

Store is similar.

Executing Code

Recall: a program is stored in RAM away from the CPU.

How does the CPU know where the next instruction to execute is?

- A special register called the Program Counter (PC) stores the memory address of the next instruction to execute.
- Instruction Register (IR) holds current instruction.
- A program needs a starting point. By convention, we guarantee that a specific address (such as 0) contains code.

PC holds the address of the next instruction while the current instruction is executing.

A program called a *loader* loads a program into memory and sets the PC to the address of the first instruction.

CS241 uses `mips.twoints` and `mips.array` to load programs into memory address 0.

`mips.twoints` inputs two integer values that will be stored in `$1` and `$2`.

`mips.array` inputs an array of integers where the base address is stored in `$1` and length in `$2`.

We will discuss other possible loaders later.

Fetch-Execute Cycle

The only program your computer really runs is:

```
PC <- 0
loop
    IR <- MEM[PC]    // MEM[]: memory, RAM
    PC <- PC + 4
    Decode and Execute instruction in IR
end loop
```

Ending a Program

Recall: A loader program, loaded your program into RAM and set the PC to start its execution.

The fetch-execute cycle keeps on going so what should you do when your program ends?

Ending a Program

Recall: A loader program, loaded your program into RAM and set the PC to start its execution.

The fetch-execute cycle keeps on going so what should you do when your program ends?

- When your program ends, it should return control back to the loader.

Where is the address of the next instruction stored?

Ending a Program

Recall: A loader program, loaded your program into RAM and set the PC to start its execution.

The fetch-execute cycle keeps on going so what should you do when your program ends?

- When your program ends, it should return control back to the loader.

Where is the address of the next instruction stored? **PC**

What memory address do we need to set PC to?

Ending a Program

Recall: A loader program, loaded your program into RAM and set the PC to start its execution.

The fetch-execute cycle keeps on going so what should you do when your program ends?

- When your program ends, it should return control back to the loader.

Where is the address of the next instruction stored? **PC**

What memory address do we need to set PC to?

Address of next instruction of the loader program.

Which memory address is that???

Which memory address is that???

The memory address you want is in $31.

Remember $31 is special.

Which memory address is that???

The memory address you want is in $31.

Remember $31 is special.

> The loader will set $31 with the address of **its** next instruction.

To set the PC $\leftarrow$ $31, use the Jump Register instruction `jr $31`

Example: Write a program that adds the values in registers $1 and $2, stores the sum in $3 and returns.

```
add $3, $1, $2
jr $31
```

Exercise: convert to binary and hexadecimal.

Getting Started on A1

```
241:> source /u/cs241/setup
241:> cs241.wordasm < add1_2.hex > add1_2.mips
241:> mips.twoints add1_2.mips
```

Address	Memory (RAM)	Assembly
0x0000	0000 0000 0010 0010 0001 1000 0010 0000	add $3, $1, $2
0x0004	0000 0011 1110 0000 0000 0000 0000 1000	jr $31
0x0008	...	

```
Enter value for register 1:  1
Enter value for register 2:  2
Running MIPS program.
MIPS program completed normally.
$01 = 0x00000001 $02 = 0x00000002 $03 = 0x00000003
```