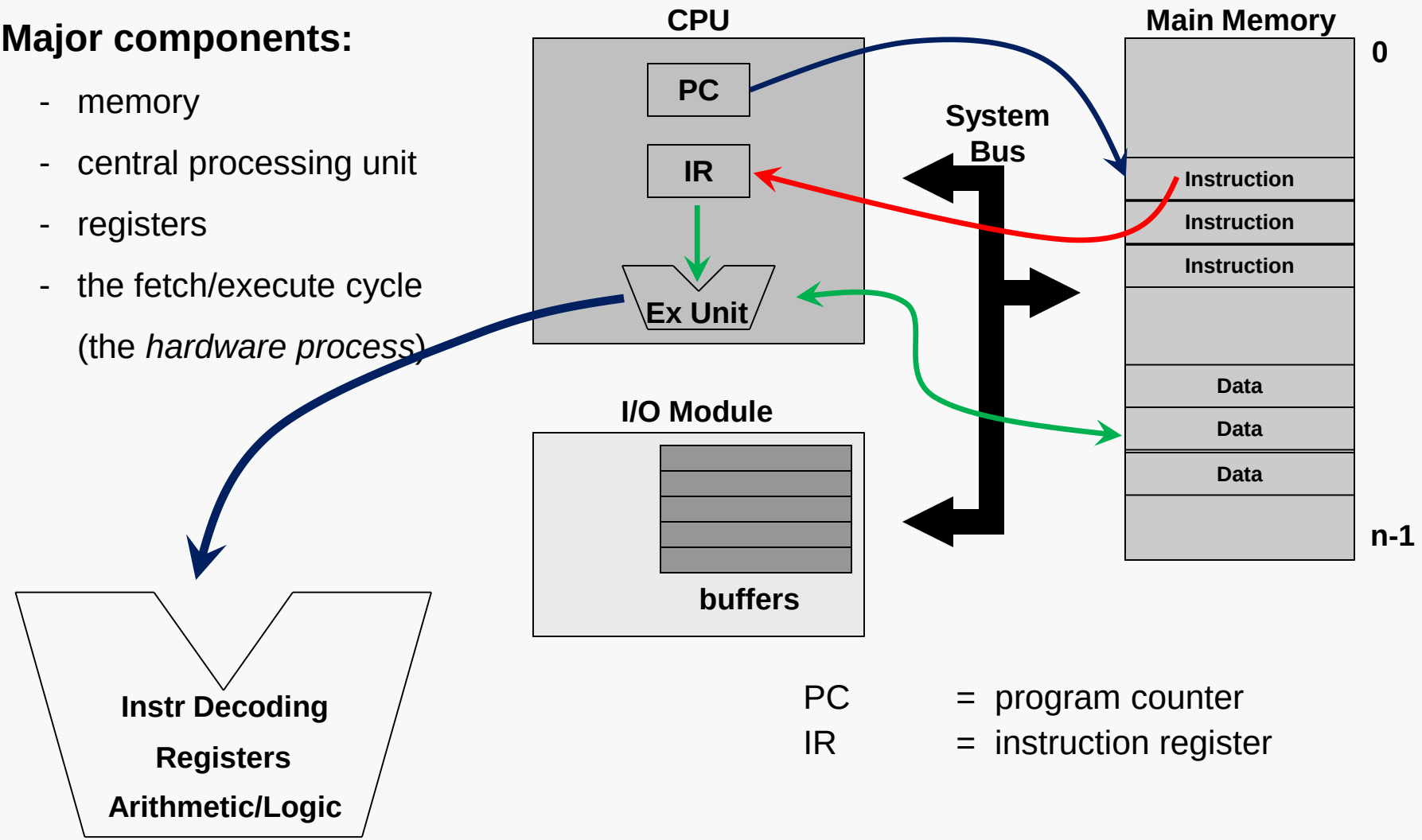
Hardware Level Organization

Major components:

- memory
- central processing unit
- registers
- the fetch/execute cycle

(the *hardware process*)



PC = program counter
IR = instruction register

Central Processing Unit

Control

- decodes instructions and manages CPU's internal resources

Registers

- general-purpose registers available to user processes
- special-purpose registers directly managed in fetch/execute cycle
- other registers may be reserved for use of operating system
- very fast and expensive (relative to memory)
- hold all operands and results of arithmetic instructions (on RISC systems)
- save bits in instruction representation

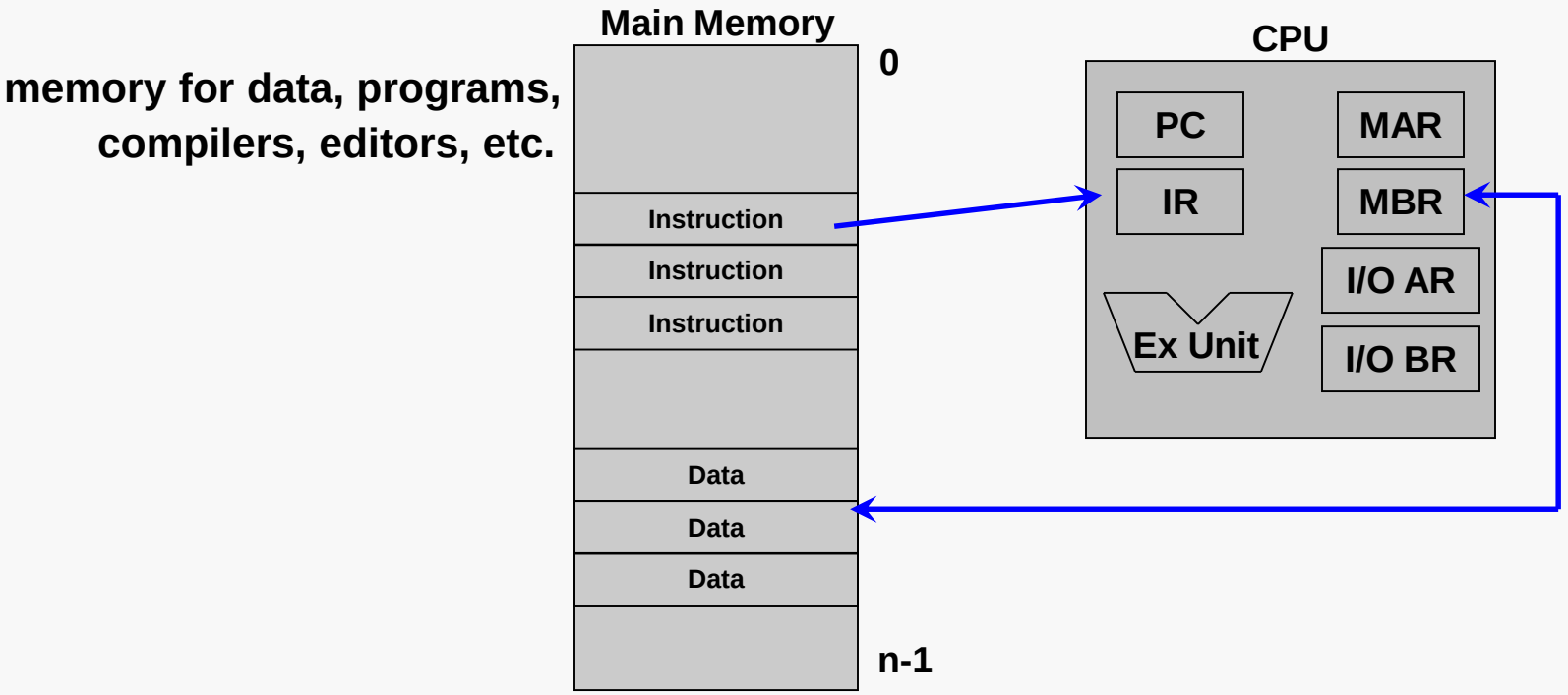
Data path or arithmetic/logic unit (ALU)

- operates on data

Stored Program Concept

Instructions are collections of bits

Programs are stored in memory, to be read or written just like data



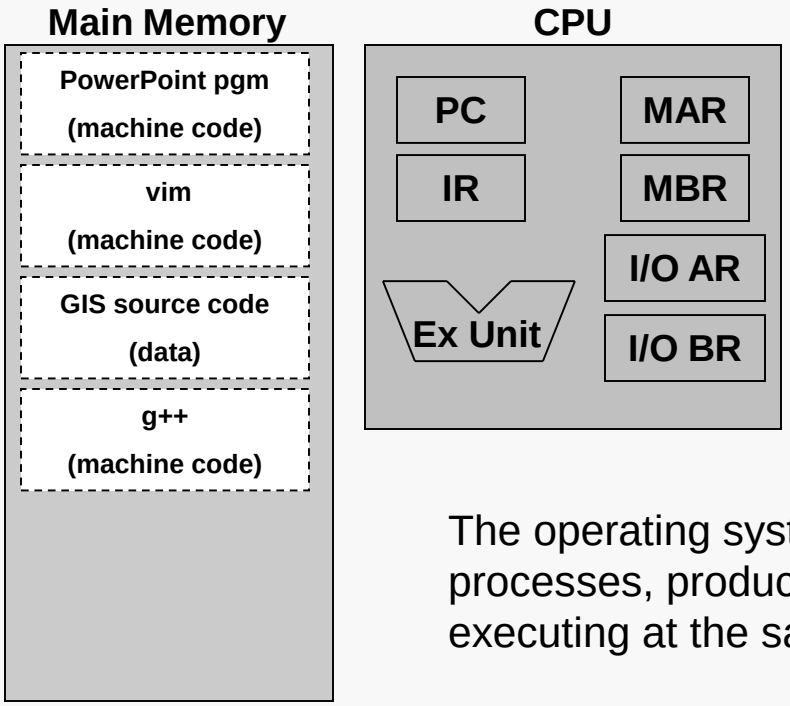
Fetch & Execute Cycle

- Instructions are fetched and put into a special register
- Bits in the register "control" the subsequent actions
- Fetch the “next” instruction and continue

Stored Program Concept

Of course, on most systems several programs will be stored in memory at any given time.

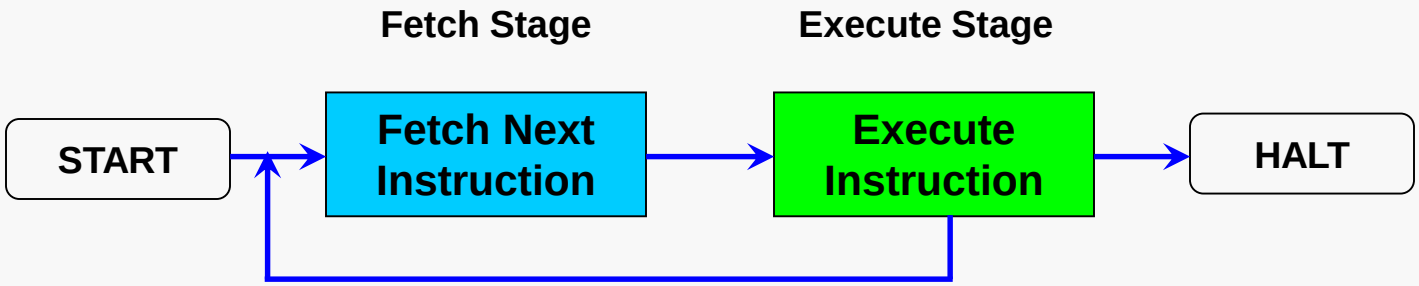
On most contemporary systems instructions of only one of those will be executed at any given instant.



The operating system will rapidly switch among the eligible processes, producing the illusion that several programs are executing at the same time.

Fetch/Execute Cycle

Sometimes called the hardware process… executes continuously.



Steps:

- fetch an instruction from memory to the *instruction register*
- increment the *program counter* register (by the instruction length)
- decode the instruction (in the control unit)
- fetch operands, if any, usually from registers
- perform the operation (in the data path); this may modify the PC register
- store the results, usually to registers

Machine Language

But, how is all of this driven?

Machine language:

- registers store collections of bits
- all data and instructions must be encoded as collections of bits (*binary*)
- bits are represented as electrical charges (more or less)
- control logic and arithmetic operations are implemented as circuits, which are driven by the movement of electrical charges
- so, the instructions directly manipulate the underlying hardware (cool, huh?)

The collection of all valid binary instructions is known as the *machine language*.

- what's valid depends on the design of the hardware, especially the control circuitry
- must be formally specified
- machine language is not human-friendly

Assembly Language

More human-friendly syntax:

- expressed in text, not in binary
- instructions are identified by (more-or-less) mnemonic names
- instruction operands may include registers, memory locations, or…

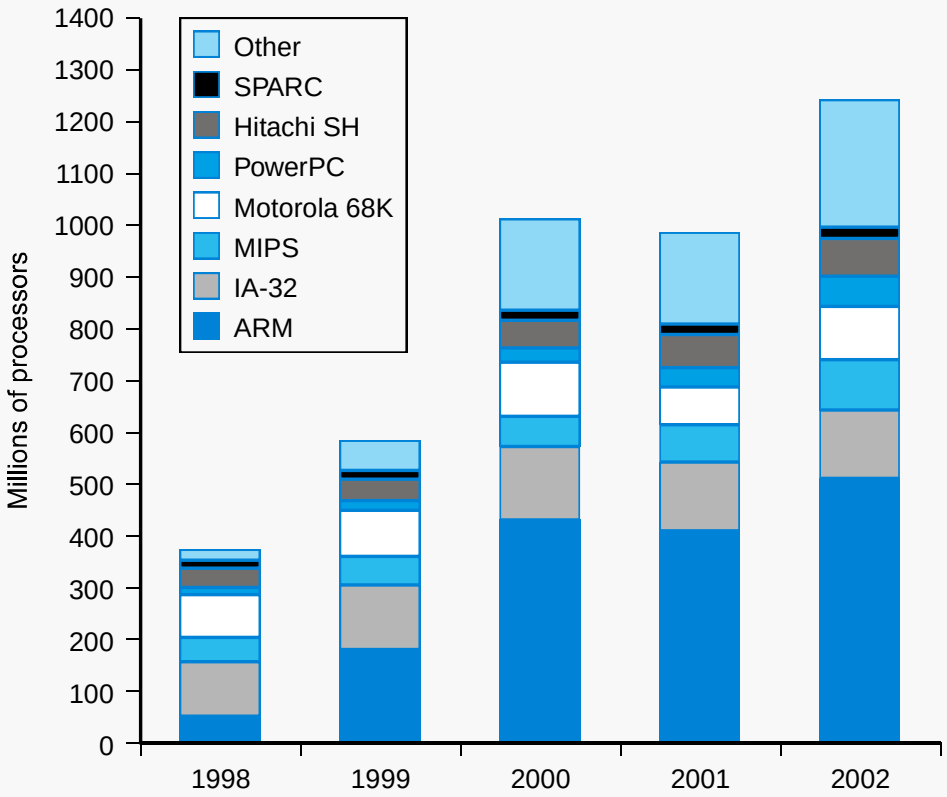
Aspects of assembly language:

- unlike high-level languages, each instruction is extremely simple, so assembly language programs are much longer than corresponding high-level language programs
- assembly language must be translated into machine language in order to be executed
- assembly language is not usually any more portable across different hardware platforms that is machine language
- most assembly languages are quite similar… from a certain point of view

MIPS ISA

We'll be working with the MIPS instruction set architecture (ISA)

- similar to other architectures developed since the 1980's
- almost 100 million MIPS processors manufactured in 2002
- used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …



MIPS Registers

Registers

- 32 32-bit general-purpose registers, referred to as $0, $1, …, $31
- 32 32-bit floating-point registers, referred to as $f0, $f1, … $f31
- 16 64-bit floating-point registers, referred to as $f0, $f2, … $f30
- conventions govern the use of the general registers

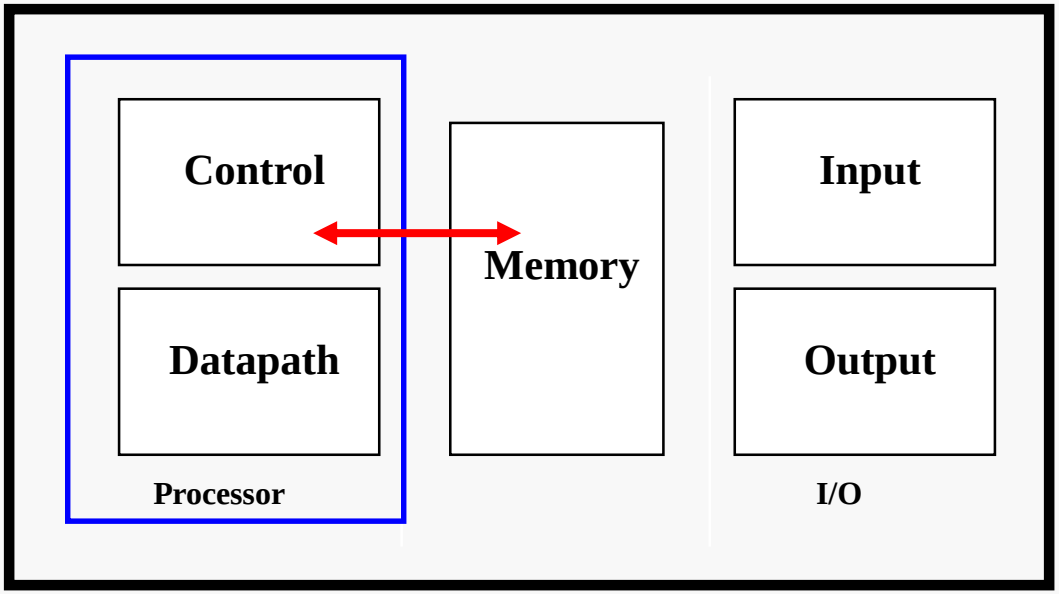
We will, for now, adopt the view that the underlying computer is a “black box” that understands MIPS machine language.

Registers vs. Memory

Operands to arithmetic and logical instructions must be registers or immediates.

Compiler associates variables with registers

What about programs with lots of variables?



MIPS Assembly Language

We will study the MIPS assembly language as an exemplar of the concept.

MIPS assembly instructions each consist of a single token specifying the command to be carried out, and zero or more operation parameters:

```
<mnemonic>  par1  par2  … parN
```

The tokens are separated by commas. Indentation is insignificant to the assembler, but is certainly significant to the human reader.

MIPS command tokens are short and mnemonic (in principle). For example:

```
add   lw    sw    jr
```

MIPS Assembly Language

MIPS operands include:

- hardware registers
- offset and base register
- literal constants (*immediate* parameters)
- labels

Of course, MIPS assembly also allows comments. Simply, all characters from a '#' character to the end of the line are considered a comment.

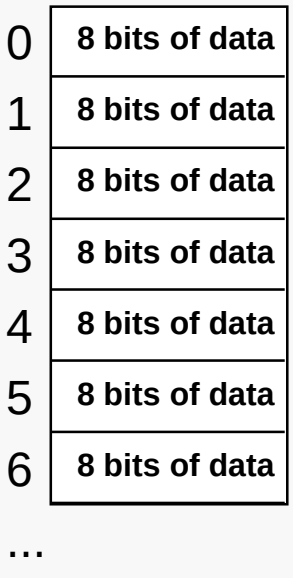
There are also some special *directives*, but those can wait...

Memory Organization

Viewed as a large, single-dimension array, with an address.

A memory address is an index into the array

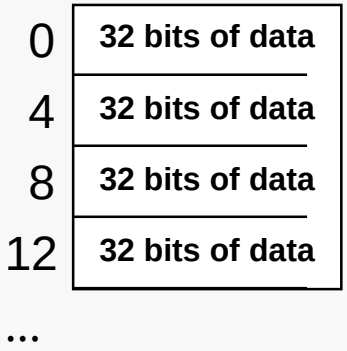
"Byte addressing" means that the index points to a byte of memory.



MIPS Memory Organization

Bytes are nice, but most data items use larger "words"

For MIPS, a *word* is 32 bits or 4 bytes.



Registers hold 32 bits of data

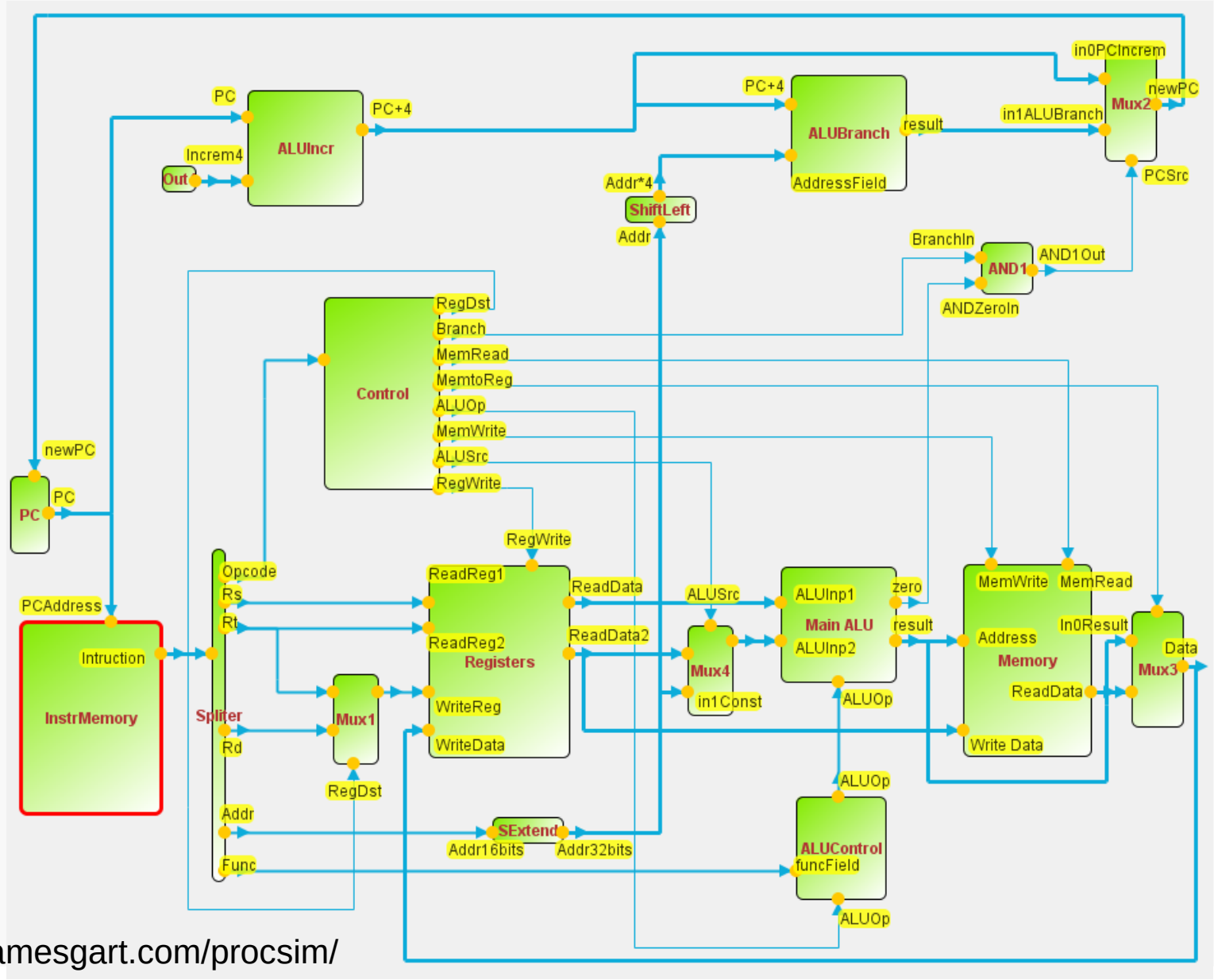
2^{32} bytes with byte addresses from 0 to $2^{32} - 1$

2^{30} words with byte addresses 0, 4, 8, ... $2^{32} - 4$

Words are *aligned*, that is, each has an address that is a multiple of 4.

MIPS can be either *big-endian* (that is, the address of each word is the address of the “left-most” byte of the word) or *little-endian*. This is important when viewing the contents of memory.

Simple MIPS Overview



http://jamesgart.com/procsim/