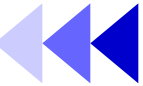
Dept. Electronics Engineering, National Chiao Tung University

DEE 1050 Computer Organization Lecture 3 Instructions: Language of the Computer

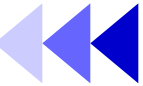
Dr. Tian-Sheuan Chang
tschang@twins.ee.nctu.edu.tw
Dept. Electronics Engineering
National Chiao-Tung University

Outline



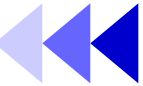
- Part I. Introduction
- Part II. Case study with MIPS instructions
- Part III. Tool chain

Outline



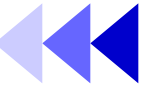
- Operations
- Operands
- Control flow
- MIPS addressing mode

Introduction



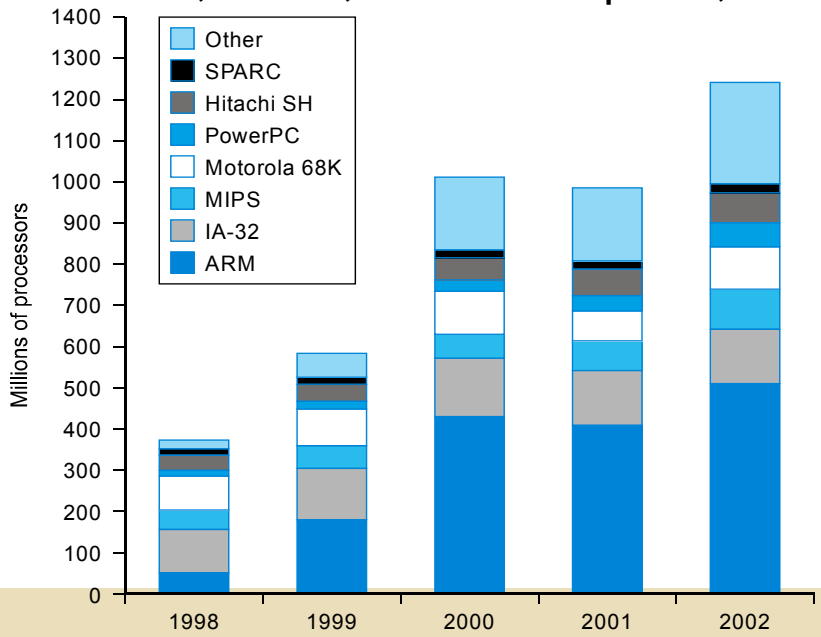
- Computer language
  - Words: instructions
  - Vocabulary: instruction set
  - Similar for all, like regional dialect ?
- Design **goal** of computer language
  - To find a language that makes it easy to build the **hardware** and the **compiler** while maximizing **performance** and minimizing **cost**
- Reading list
  - COD3E: ch. 2

Instructions: Difference with HLL

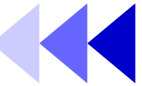


- Language of the Machine
  - More **primitive** than higher level languages
    e.g., no sophisticated control flow
  - Very **restrictive**
    e.g., MIPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  - 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, …



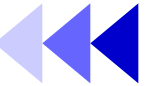
How to Design the Instructions?



- Operations
  - Arithmetic
  - Logical
  - **=> *Datapath***
- Operands
  - **=> *Datapath***
- Control flow
  - Decision control
  - Procedures calls
  - **=> *Control***

```
int add5 (int a)
{
   int tmp = a + 5;
   return tmp;
}
void main ()
{
  int a = 7;
  int c;
  if (a == 7)
     c = add5(a);
}
```

Operations: MIPS arithmetic



- Each arithmetic instructions performs only **one operation** and have **3 operands**
- Operand order is fixed (destination first)

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

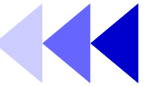
One operation

Exact three operands

comments

"The natural number of operands for an operation like addition is *three*…requiring every instruction to have exactly three operands, no more and no less, conforms to the **philosophy of keeping the hardware simple**"

MIPS arithmetic



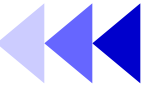
- **Design Principle 1: simplicity favors regularity**.
- Of course this complicates some things...

C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

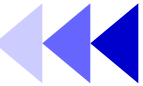
Q. Will variable no. of operands be faster?
how about four or more operands in one instructions

MIPS Logical Operations (Section 2.5)



- Why logical operations
  - Useful to operate on fields of bit or individual bits
- Q. Can some **multiply by 2^i ? Divide by 2^i ? Invert?**
- Q. Why "*NOT*" maps to "*nor*"?
  - *A NOR 0 = NOT (A OR 0) = NOT (A)*
- Q. Why not "*nori*" (nor immediate)?
  - Constant are rare for for NOR

operations	c operators	mips
shift left	<<	sll
shift right	>>	srl
bit-by-bit and	&	and, andi
bit-by-bit or	|	or, ori
bit-by-bit not	~	nor

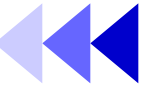


- Intended to be blank

Operands of the Computer Hardware (Section 2.3)

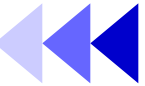
- Difference with HLL like C
  - Limited number, why ?
  - Operands are restricted to hardware-built registers
  - Registers are **primitive and visible** to programmer
- MIPS operands
  - Operands must be **registers**, only **32** registers provided
  - Each register contains 32 bits
  - Why 32?
- **Design Principle 2: smaller is faster**.

Operand Type



- 3 Types
  - Register operands
    - All arithmetic operations are in the register operands
  - Memory operands
    - Array or structure
    - Only load/store can access memory
  - Constant or immediate operands
    - Small value will be in the instruction
    - Large value will be stored separately

Register Operand Example

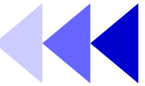


- Register representation
  - $**, in MIPS
    - $s0, $s1.. Registers corresponding to the variables of C programs
    - $t0, $t1… temporary registers need to compile the program
  - (this might be different in other assembly language)

Translate the following C program into MIPS

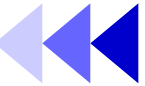
```
f = (g + h) – ( i + j);
Assume f, g, h, i, j uses $s0, .. $s4

Add $t0, $s1, $s2
Add $t1, $s3, $s4
Sub $s0, $t0 - $t1
```

HW/SW IF: How Compiler Use Registers



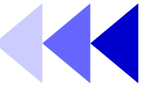
- Problem: more variables than available registers
- Solution
  - Keep the most **frequently used variables** in registers
  - Place the rest in memory (called *spilling registers*), use load and store to move variables between registers and memory
  - *Why?*
    - *Register is faster but its size is small*
    - *Compiler must use register efficiently*

Memory Operands: Array and Structures



- Data are stored in memory
- “data transfer instructions”
  - Transfer data between memory and registers
  - Load `lw`: move data from memory to a register
  - Store `st`: move data from a register to memory

Array Example



- Load format
  - `lw register names, const offset(base register)`

```
g = h + a[8]
```

assume g, h => $s1, $s2

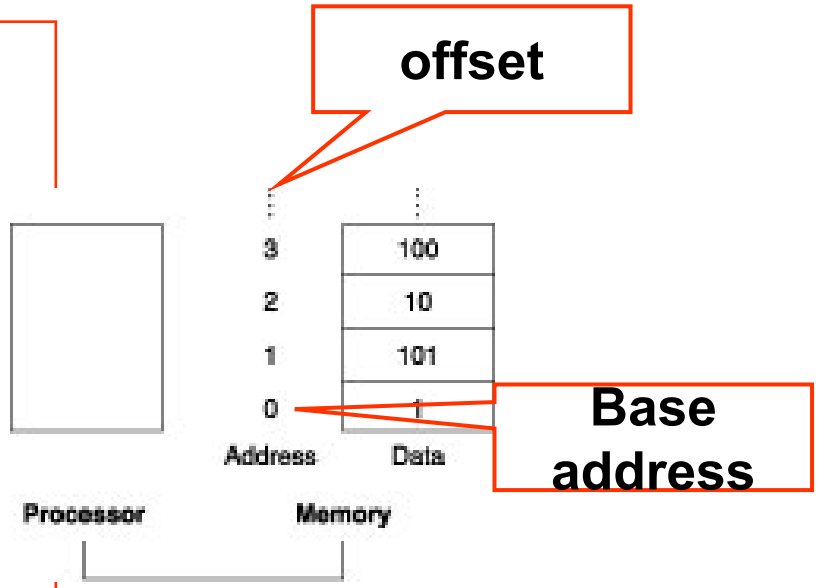
base address => $s3

```
lw $t0, 8($s3)   #lw: load word
add $s1, $2, $t0
```

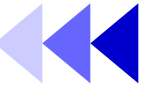
If a[12] = h + a[8]

Add one more instructions

```
sw 12[$s3], $s1
```



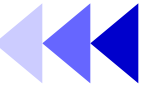
Memory and Data Sizes



So far, we've only talked about uniform data sizes. Actual data come in many different sizes:

- Single bits: ("boolean" values, true or false)
- **Bytes (8 bits):** Characters (ASCII), very small integers
- **Halfwords (16 bits):** Characters (Unicode), short integers
- **Words (32 bits):** Long integers, floating-point (FP) numbers
- Double-words (64 bits): Very long integers, double-precision FP
- Quad-words (128 bits): Quad-precision floating-point numbers

Different Data Sizes

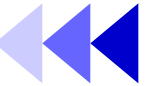


How do we handle different data sizes?

- Pick one size to be the unit stored in a single address
- Store larger datum in a set of contiguous memory locations
- Store smaller datum in one location; use shift & mask ops

Today, almost all machines (including MIPS) are "**byte-addressable**" – each addressable location in memory holds 8 bits.

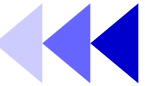
Memory Organization – Byte Addressing



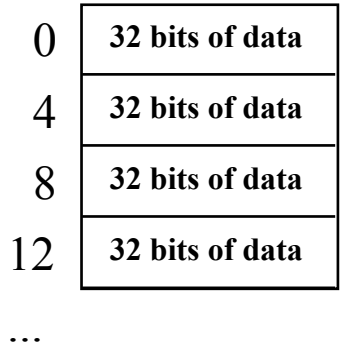
- Viewed as a large, single-dimension array, with an address.
- A memory address is an **index into the array**
- **"Byte addressing"** means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

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**
  i.e., what are the least 2 significant bits of a word address? To select the byte
- **Alignment restriction in MIPS**
  - **Words must start at addresses that are multiples of** 4

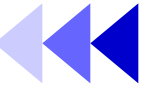
Array Example for Real MIPS Memory Address

- Code for byte addressable memory

Original	Updated
`a[12] = h + a[8]`	`a[12] = h + a[8]`
assume g, h => $s1, $s2	assume g, h => $s1, $s2
base address => $s3	base address => $s3, word data
`lw $t0, 8($s3)`	`lw $t0, 32($s3)`
`add $s1, $2, $t0`	`add $s1, $2, $t0`
`sw 12($s3), $s1`	`sw 48($s3), $s1`

Remember **arithmetic operands are registers, not memory**!
Can't write: add 48($s3), $s2, 32($s3)

Byte-Order ("Endianness")



For a multi-byte datum, which part goes in which byte?

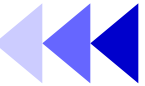
- If $1 contains 1,000,000 (F4240H) and we store it into address 80:
- On a "big-endian" machine, the **"big" end** goes into address 80

		00	0F	42	40			
…	79	80	81	82	83	84	…	

- On a "little-endian" machine, it's the other way around

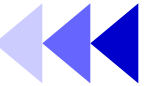
		40	42	0F	00			
…	79	80	81	82	83	84	…	

Big-Endian vs. Little-Endian



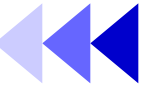
- Big-endian machines: MIPS, Sparc, 68000
- Little-endian machines: most Intel processors, Alpha, VAX
  - No real reason one is better than the other…
  - Compatibility problems transferring multi-byte data between big-endian and little-endian machines – **CAREFUL**!
- Bi-endian machines: ARM, User's choice

Registers Operands vs. Memory Operands



- Arithmetic instructions operands must be registers,
  - only 32 registers provided
  - Compiler associates variables with registers
- What about programs with lots of variables ? Like array and structures
  - Data structures are kept in memory
  - **Data transfer instructions**
    - Load: ***lw*** copy data from memory to registers
    - Store: ***sw*** copy data from registers to memory
    - How: instruction supplies the memory address

Constant or Immediate Operands



- **Small constants are used quite frequently (>50% of operands in SPEC2000 benchmark)**

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

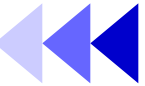
- Solutions? *Why not?*
  - put 'constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:

```
addi $s1, $s1, 5
slti $s2, $s1, 10
andi $s1, $s1, 6
ori $s1, $s1, 4
```

- **Design Principle 3: Make the common case fast**.
- *Q: why only "addi" and no "subi"*
  - *Negative constants*

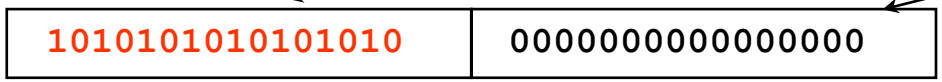
How about larger constants?



- We'd like to be able to load a 32 bit constant into a register
- Must **use two instructions**, new "load upper immediate" instruction

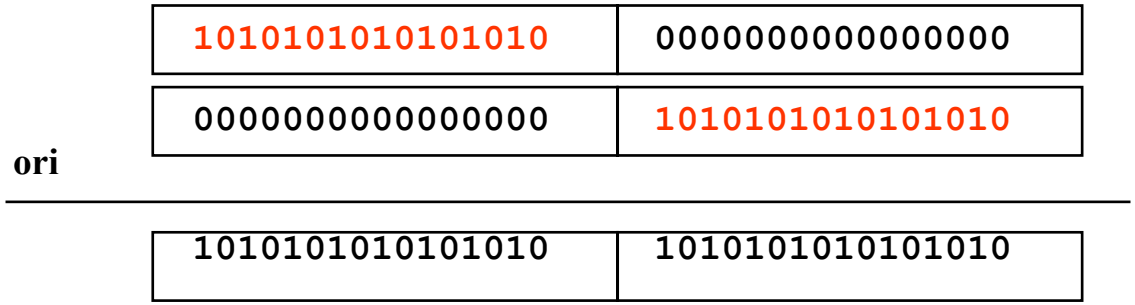
```
lui $t0, 1010101010101010
```

filled with zeros

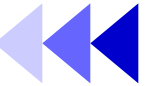


- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



So far



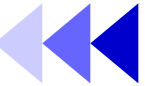
MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp(29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.
2^{30} memory words	Memory [0], Memory [4],…, Memory[42949672920	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled register, such as those saved on procedure calls.

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
Data transfer	load word	lw $s1,100 ($s2)	&s1 = Memory [$s2 + 100]	Data from memory to register
	store word	sw $s1,100 ($s2)	Memory [$s2 + 100] = $s1	Data from register to memory
Conditional branch	branch on equal	beq $s1, $s2, L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1, $s2, L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than: for beq, bne
Unconditional 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 1000	For procedure call

INFO: *MIPS Registers*

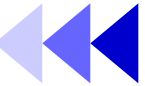


- 32 regs with R0 = 0
- Reserved registers : R1, R26, R27.
- Special usage:
  - R28: pointer to global area
  - R29: stack pointer
  - R30: frame pointer
  - R31: return address

INFO: Standard Register Conventions

- The 32 integer registers in the MIPS are “general-purpose” – any can be used as an operand or result of an arithmetic op
- But **making** different pieces of **software** work together is **easier** if certain conventions are followed concerning which registers are to be used for what purposes.
- These conventions are usually suggested by the vendor and supported by the compilers

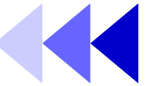
INFO: MIPS Registers and Usage Convention



Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

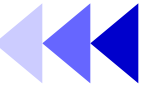
Register 1 ($at) reserved for assembler, 26-27 for operating system

INFO: MIPS Registers and Usage Convention



Register name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and results of a function
v1	3	Expression evaluation and results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved aross call)
t1	9	Temporary (not preserved aross call)
t2	10	Temporary (not preserved aross call)
t3	11	Temporary (not preserved aross call)
t4	12	Temporary (not preserved aross call)
t5	13	Temporary (not preserved aross call)
t6	14	Temporary (not preserved aross call)
t7	15	Temporary (not preserved aross call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved aross call)
t9	25	Temporary (not preserved aross call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Our First Example

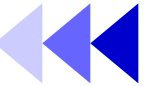


- Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
     temp = v[k]
     v[k] = v[k+1];
     v[k+1] = temp;
}
```

```
swap:
     muli $2, $5, 4
     add $2, $4, $2
     lw $15, 0($2)
     lw $16, 4($2)
     sw $16, 0($2)
     sw $15, 4($2)
     jr $31
```

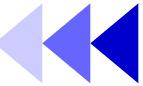
So far we've learned:



- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

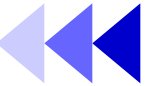
- Instruction — Meaning

```
add $s1, $s2, $s3        $s1 = $s2 + $s3
sub $s1, $s2, $s3        $s1 = $s2 - $s3
lw $s1, 100($s2)         $s1 = Memory[$s2+100]
sw $s1, 100($s2)         Memory[$s2+100] = $s1
```



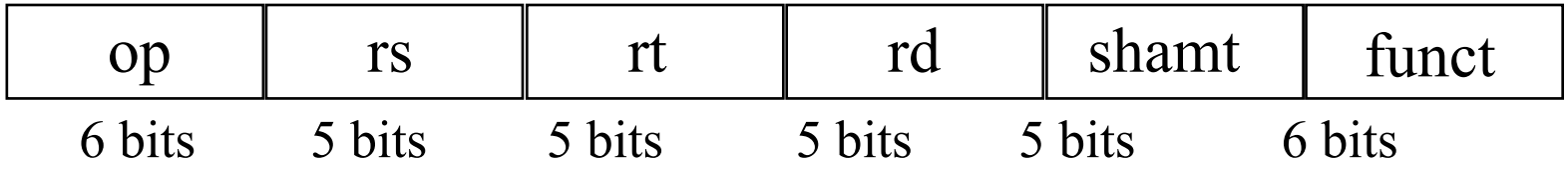
- Intended to be blank

Instruction Format (Section 2.4)



MIPS: 32-bit instruction and data

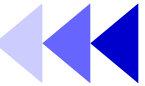
MIPS fields are given names to make them easier to discuss: (R-type)



Here is the meaning of each name of the fields in MIPS instructions:

- *op*: **operation** of the instruction, called the **opcode**
- *rs*: the first register source operand
- *rt*: the second register source operand
- *rd*: the register destination operand; it gets the result of the operation
- *shamt*: shift amount
- *funct*: function; this field selects the **variant** of the operation in the op field called **function code**

Instruction Format : Example

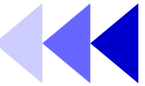


- **Instructions**, like registers and words of data, are also **32 bits long**
  - Example: `add $t1, $s1, $s2`
  - registers have numbers, `$t1=9, $s1=17, $s2=18`
    why?
- Instruction Format:

000000	10001	10010	01000	00000	100000

op	rs	rt	rd	shamt	funct

What if Longer Field is Required?



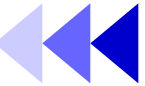
- Consider the load-word and store-word instructions
  - Load word: two registers and a constant
  - **Constant < 32** if any above 5-bit fields is used
  - What would the regularity principle have us do?
  - **Principle 4: Good design demands a compromise**
- Introduce a **new type of instruction format**
  - **I-type** for immediate and data transfer instructions
  - other format was **R-type** for register
- Example: `lw $t0, 32($s2)`

35	18	9	32

op	rs	rt	16 bit number

- Where's the compromise?
  - Keep instruction the same length with different formats
  - Keep the formats similar

Data Transfer Instructions



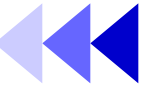
- I-type (base + 16 bit offsets)

lw t0, 8 ($s3) --- # Temporary reg t0 gets A[*8*]

Note: s3 stores the start address of A

Also, rs is the base register, rt stores the destination register.

Complete MIPS Instruction Formats



R-Format

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Format

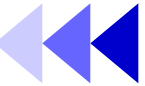
op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

J-Format

op	address
6 bits	26 bits

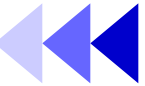
Simple and regular format

Fields in MIPS Instructions

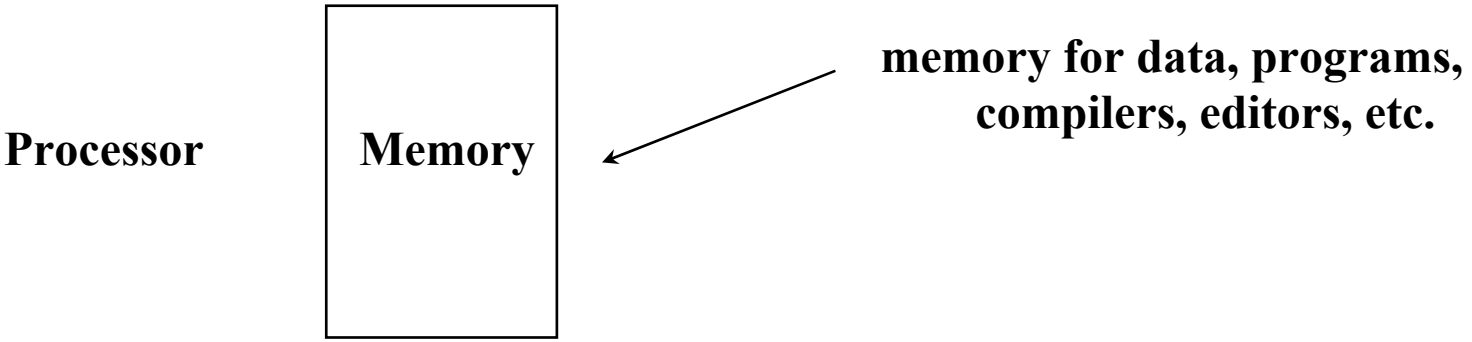


- **op**: Specifies the operation; tells which format to use
- **rs**: First source register
- **rt**: second source register (or dest. For load)
- **rd**: Destination register
- **shamt**: Shift amount
- **funct**: Further elaboration on opcode
- **address**: immediate constant, displacement, or branch target

BIG PICTURE: *Stored Program Concept*



- Instructions are represented as **numbers**
- **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
- Consequence
  - **Binary compatibility** due to number representation