Computer Organization and Architecture

MIPS Architecture

Lecture: 01

Fall, 2020

MIPS

- Execution time =

  instructions per program * cycles per instruction * seconds per cycle

- MIPS is implementation of a RISC architecture

- MIPS R2000 ISA
  - Designed for use with high-level programming languages
    - » small set of instructions and addressing modes, easy for compilers
  - Minimize/balance amount of work (computation and data flow) per instruction
    - » allows for parallel execution
  - Load-store machine
    - » large register set, minimize main memory access
  - fixed instruction width (32-bits), small set of uniform instruction encodings
    - » minimize control complexity, allow for more registers

Instruction Sets

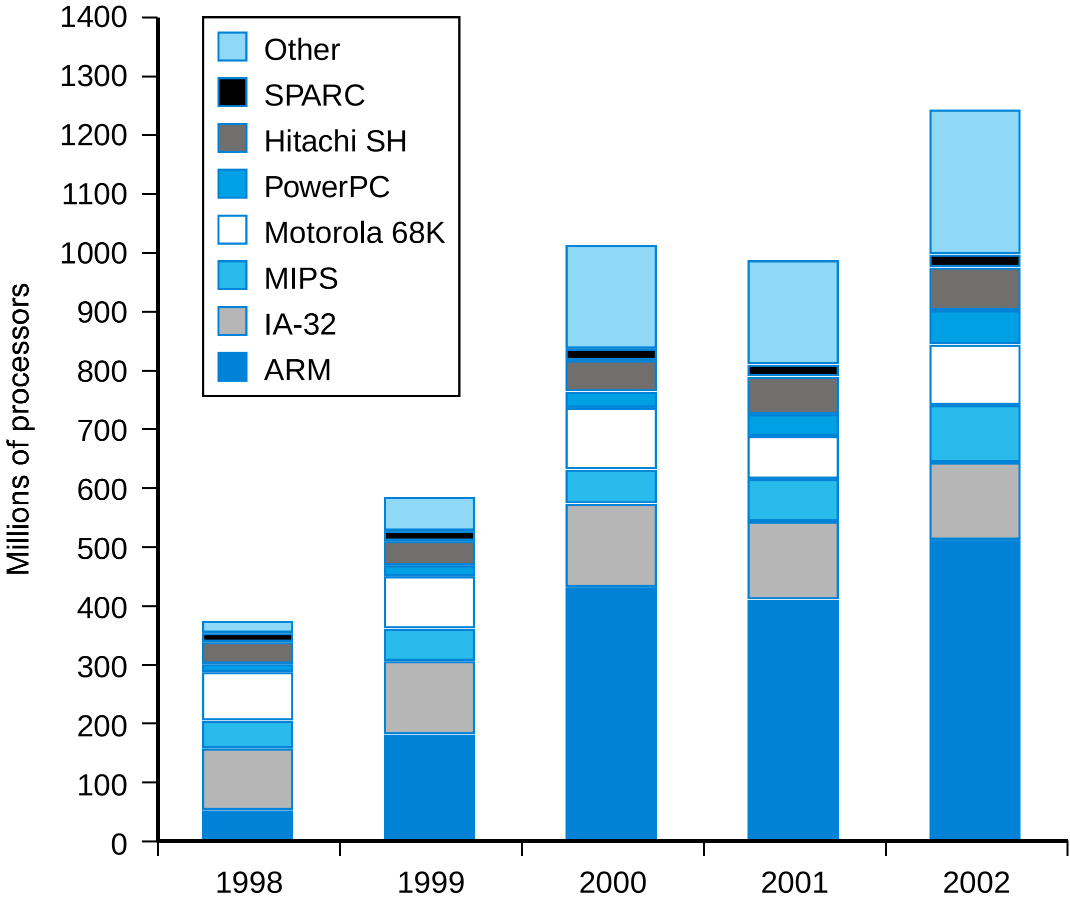
- Instruction Set Architecture (ISA)
  - Usually defines a “family” of microprocessors
    - » Examples: Intel x86 (IA32), Sun Sparc, DEC Alpha, IBM/360, IBM PowerPC, M68K, DEC VAX
  - Formally, it defines the interface between a user and a microprocessor

- ISA includes:
  - Instruction set
  - Rules for using instructions
    - » Mnemonics, functionality, addressing modes
  - Instruction encoding

- ISA is a form of *abstraction*
  - Low-level details of microprocessor are “invisible” to user

MIPS

- **MIPS: Microprocessor without Interlocked Pipeline Stages**
- **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, …**



MIPS Design Principles

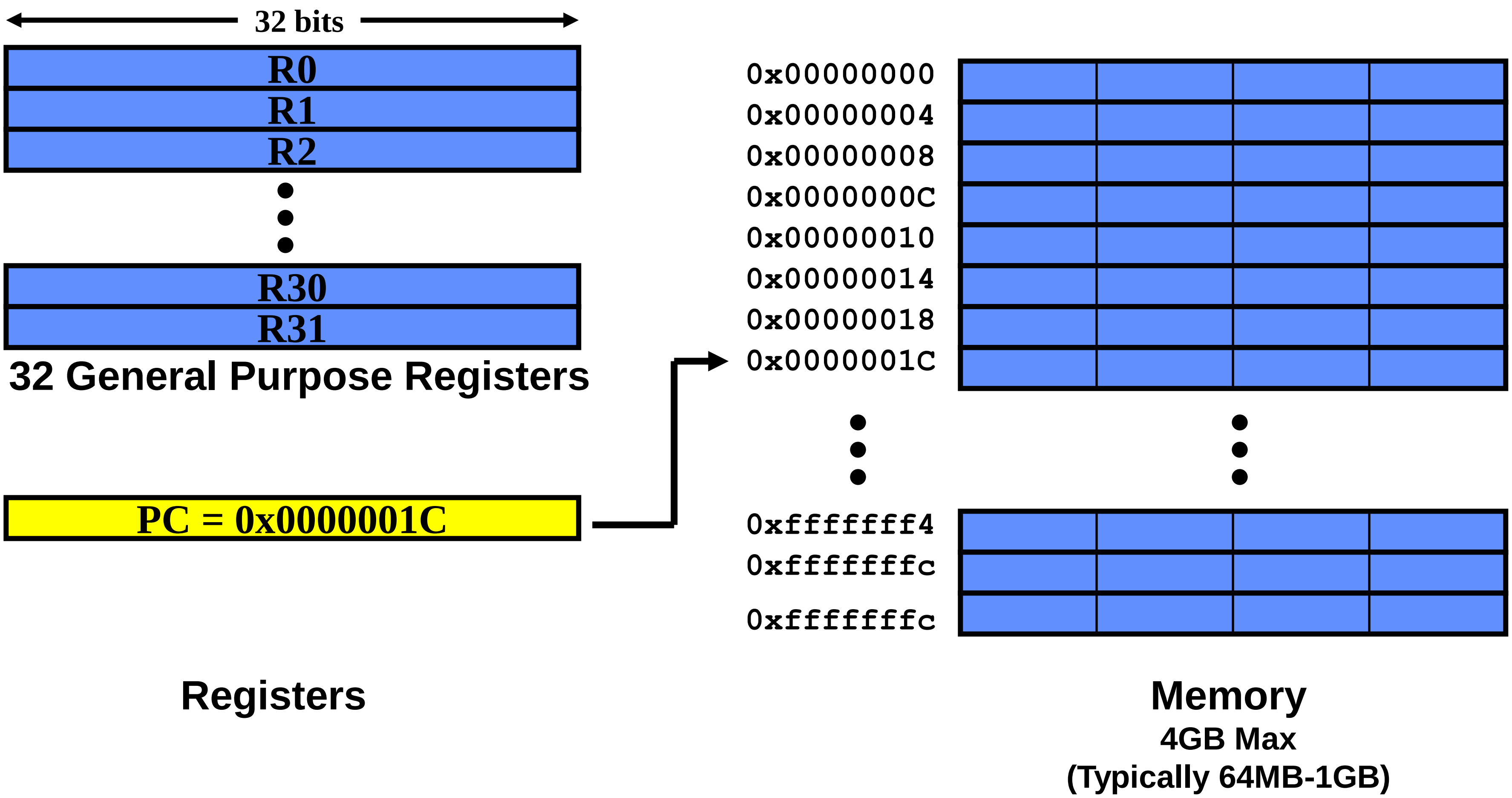
1. **Simplicity Favors Regularity**
   - Keep all instructions a single size
   - Always require three register operands in arithmetic instructions

2. **Smaller is Faster**
   - Has only 32 registers rater than many more

3. **Good Design Makes Good Compromises**
   - Comprise between providing larger addresses and constants instruction and keeping instruction the same length

4. **Make the Common Case Fast**
   - PC-relative addressing for conditional branches
   - Immediate addressing for constant operands

MIPS Registers and Memory



MIPS Registers and Usage

Name	Register number	Usage
`$zero`	0	the constant value 0
`$at`	1	reserved for assembler
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporary registers
`$s0-$s7`	16-23	saved registers
`$t8-$t9`	24-25	more temporary registers
`$k0-$k1`	26-27	reserved for Operating System kernel
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

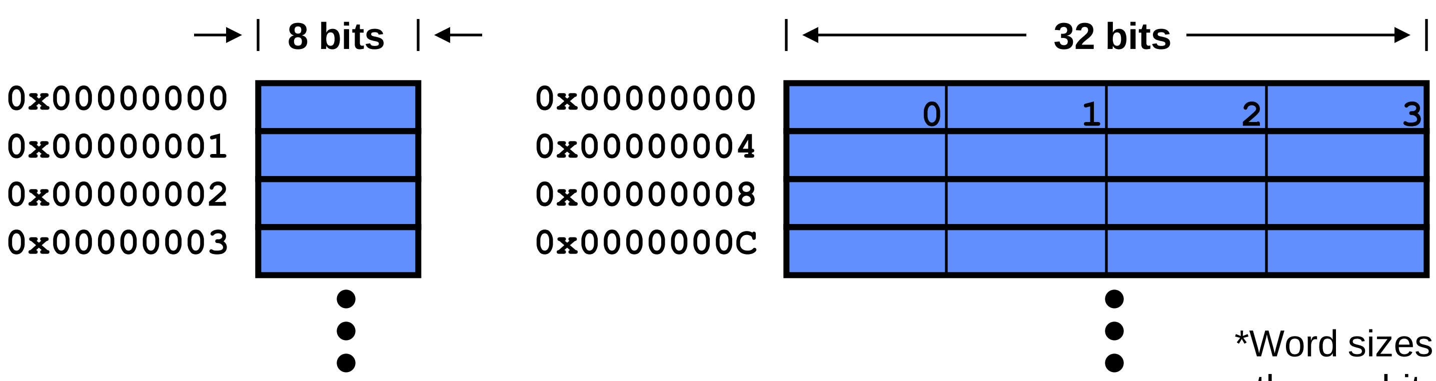
Each register can be referred to by number or name.

More about MIPS Memory Organization

- **Two** views of memory:
  - 2^{32} **bytes** with addresses 0, 1, 2, …, 2^{32}-1
  - 2^{30} 4-byte **words*** with addresses 0, 4, 8, …, 2^{32}-4
- **Both views use byte addresses**

Not all architectures require this

- **Word address must be multiple of 4 (aligned)**



*Word sizes vary in other architectures

MIPS Instruction Types

- **Arithmetic & Logical** - manipulate data in registers

add $s1, $s2, $s3	$s1 = $s2 + $s3
or $s3, $s4, $s5	$s3 = $s4 OR $s5

- **Data Transfer** - move register data to/from memory

lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]
sw $s1, 100($s2)	Memory[$s2 + 100] = $s1

- **Branch** - alter program flow

beq $s1, $s2, 25	if ($s1==$s1) PC = PC + 4 + 4*25

Name	Parameters	Description
`.align`	*n*	Align the next item on the next 2^n-byte boundary. `.align 0` turns off automatic alignment.
`.ascii`	*str*	Assemble the given string in memory. Do not null-terminate.
`.asciiz`	*str*	Assemble the given string in memory. Do null-terminate.
`.byte`	*byte1* $\cdots$ *byteN*	Assemble the given bytes (8-bit integers).
`.half`	*half1* $\cdots$ *halfN*	Assemble the given halfwords (16-bit integers).
`.space`	*size*	Allocate n bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
`.word`	*word1* $\cdots$ *wordN*	Assemble the given words (32-bit integers).

Syscall

Service	Code	Arguments	Result
print_int	1	$a0	*none*
print_float	2	$f12	*none*
print_double	3	$f12	*none*
print_string	4	$a0	*none*
read_int	5	*none*	$v0
read_float	6	*none*	$f0
read_double	7	*none*	$f0
read_string	8	$a0 (address), $a1 (length)	*none*
sbrk	9	$a0 (length)	$v0
exit	10	*none*	*none*

-
- **main:**
- **## Get first number from user, put into $t0.**
- **li $v0, 5 # load syscall read_int into $v0.**
- **syscall # make the syscall.**
- **move $t0, $v0 # move the number read into $t0.**
-
- **## Get second number from user, put into $t1.**
- **li $v0, 5 # load syscall read_int into $v0.**
- **syscall # make the syscall.**
- **move $t1, $v0 # move the number read into $t1.**
- **add $t2, $t0, $t1 # compute the sum.**
-
- **## Print out $t2.**
- **move $a0, $t2 # move the number to print into $a0.**
- **li $v0, 1 # load syscall print_int into $v0.**
- **syscall # make the syscall.**
-
- **li $v0, 10 # syscall code 10 is for exit.**
- **syscall # make the syscall.**

For print string

- data
- str: .asciiz “the answer = “
- .text
-
- li $v0, 4 # system call code for print_str
- la $a0, str # address of string to print
- syscall # print the string
-
- li $v0, 1 # system call code for print_int
- li $a0, 5 # integer to print
- syscall # print it
-

MIPS Arithmetic & Logical Instructions

- **Instruction usage (assembly)**

add dest, src1, src2	dest=src1 + src2
sub dest, src1, src2	dest=src1 - src2
and dest, src1, src2	dest=src1 AND src2

- **Instruction characteristics**
  - Always 3 operands: destination + 2 sources
  - Operand order is fixed
  - Operands are always general purpose registers
- **Design Principles:**
  - Design Principle 1: Simplicity favors regularity
  - Design Principle 2: Smaller is faster

Logical instruction

- **and** **and $1,$2,$3** **$1 = $2 & $3** **Bitwise AND**
- **or** **or $1,$2,$3** **$1 = $2 | $3** **Bitwise OR**
- **xor** **xor $1,$2,$3** **$1 = $2 ??$3** **Bitwise XOR**
- **nor** **nor $1,$2,$3** **$1 = ~($2 | $3)** **Bitwise NOR**
- **and immediate** **andi $1,$2,10** **$1 = $2 & 10** **Bitwise AND reg, const**
- **or immediate** **ori $1,$2,10** **$1 = $2 | 10** **Bitwise OR reg, const**
- **xor immediate** **xori $1, $2,10** **$1 = ~$2 &~10** **Bitwise XOR reg, const**
- **shift left logical** **sll $1,$2,10** **$1 = $2 << 10** **Shift left by constant**
- **shift right logical** **srl $1,$2,10** **$1 = $2 >> 10** **Shift right by constant**
- **shift right arithm.** **sra $1,$2,10** **$1 = $2 >> 10** **Shift right (sign extend**
- **shift left logical** **sllv $1,$2,$3** **$1 = $2 << $3** **Shift left by var**
- **shift right logical** **srlv $1,$2, $3** **$1 = $2 >> $3** **Shift right by var**
- **shift right arithm.** **srav $1,$2, $3** **$1 = $2 >> $3** **Shift right arith. by var**

MIPS Conditional Branch Instructions

- Conditional branches allow <u>decision making</u>

```
beq R1, R2, LABEL     if R1==R2 goto LABEL
bne R3, R4, LABEL     if R3!=R4 goto LABEL
```

- Example

C Code

```
        if (i==j) goto L1;
        f = g + h;
L1:     f = f - i;
```

Assembly

```
        beq $s3, $s4, L1
        add $s0, $s1, $s2
L1:     sub $s0, $s0, $s3
```

Example: Compiling C `if-then-else`

- **Example**

  C Code if (i==j) f = g + h;
  else f = g - h;

  Assembly

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j Exit;          # new: unconditional jump
Else:   sub $s0, $s0, $s3
Exit:
```

- **New Instruction: Unconditional jump**

```
j LABEL   # goto Label
```