

# CSE 220 Computer Organization

---

## Lecture 8

**Hussein Badr**  
**Computer Science, Stony Brook University**

**<http://www.cs.sunysb.edu/~cse220>**

# History of RISC Machines

---

- **As processors got powerful over time, the instruction set became very complex as well as large.**
- **A number of studies of complex, large programs written in high level languages revealed an interesting fact.**
- **Most (~85%) of the code consisted of simple assignments, 'if' statements, and function calls with few parameters.**
- **Complex instructions, although provided, were infrequently used by the compiler/programmer.**
- **These complex instructions slowed down the simplest instructions.**

# Central Idea

---

- **Design an instruction set with simple and only a few instructions.**
- **This set can be easily implemented in hardware such that they can be executed very fast.**
- **Some complex operation may require several instructions and could be slow.**
- **Since these are not very common we could afford the penalty in these cases.**
- **The net improvement in performance is much higher.**
- **RISC: Reduced Instruction Set Computing.**
- **CISC: Complex or Comprehensive I.S.C. Has many complex instructions which slow down simple instructions.**

# MIPS

---

- It stands for 'million instructions per second'.
- It is a unit of measuring processor speed.
- In 1980, Professor Patterson of UC Berkeley designed the RISC 1 processor.
- At the same time, Professor Hennessy of Stanford U. designed another RISC processor called MIPS. Here MIPS stood for Microprocessor without Interlocking Pipeline Stage.
- Later, these two designs led to two commercial processors SPARC and MIPS.
- SPARC was produced by Sun Microsystems.
- MIPS was bought out by Silicon Graphics, a maker of high-capability, graphics-oriented workstations.

# RISC VS CISC (1/2)

---

- **'CISC' at that time referred to VAX architecture that dominated Computer Science depts. at major Universities.**
- **But it included Intel and large IBM machines as well.**
- **Although RISC machines have an advantage over CISC machines, why haven't they won the battle ?**
- **Lots of investment had already taken place in Intel and IBM designs.**
- **Because of backward compatibility, users preferred new machines from Intel and IBM.**
- **Starting with the 486 and Pentium-I processors Intel implemented a RISC core in its processors.**

# RISC VS CISC (2/2)

Machine	Number of general purpose registers	Architectural Style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
MIPS	32	Load-store (RISC)	1985
HP PA-RISC	32	Load-store (RISC)	1986
SPARC	32	Load-store (RISC)	1987
PowerPC	32	Load-store (RISC)	1992
DEC Alpha	32	Load-store (RISC)	1992
HP/Intel IA-64	128	Load-store (EPIC)	2001
AMD64 (EMT64)	16	Register-memory	2003

FIGURE 2.20.1 The number of general-purpose registers in popular machines over the years  
(see Textbook CD, Section 2.20 (page 2.20-2 )

- **Load-Store: only load and store instructions refer to memory locations.**
- **Register-Memory: allows that one operand could be in memory.**
- **Memory-Memory: both operands could be in memory.**

# MIPS Registers

---

- In addition to the PC (and some other special registers), MIPS has 32 general purpose registers which are numbered from 0 to 31.
- Register  $n$  is designated by  $\$n$  or  $Rn$  or its assigned name.
- Register  $\$0$  has the value zero hard-wired into it.
- MIPS has established a set of conventions on the use of these registers.
- These conventions are guidelines and are not enforced by hardware.
- A program that violates these may still work, as long as it is stand-alone.
- However, it will not work with other software that follows guidelines.

# MIPS Registers (1/2)

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

**FIGURE 2.14** (textbook, p. 121) MIPS register conventions. Register 1, called \$at, is reserved for the assembler (see Section 2.12), and registers 26-27, called \$k0-\$k1, are reserved for the operating system. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

- **We are allowed to use either a register name or its number in our program.**

# MIPS Registers (2/2)

---

- MIPS also has 32 floating point registers, **FP0** to **FP31**.
- Used for floating point operands.
- Have connections to floating point (arithmetic) unit.
  
- Pairs of floating point registers is used for double precision numbers.
  - **FP0**, **FP2**, ..... **FP30** are for double precision.
  - **FP0** is actually {**FP0** & **FP1**}, and **FP30** is actually {**FP30** & **FP31**}, together.
  
- MIPS has some other registers.
  - **HI** & **LO** → used in multiplication.
  - **Status**, **EPC**, **Cause**, **BadVaddr** are used in errors and exception handling.

# MIPS Memory Layout

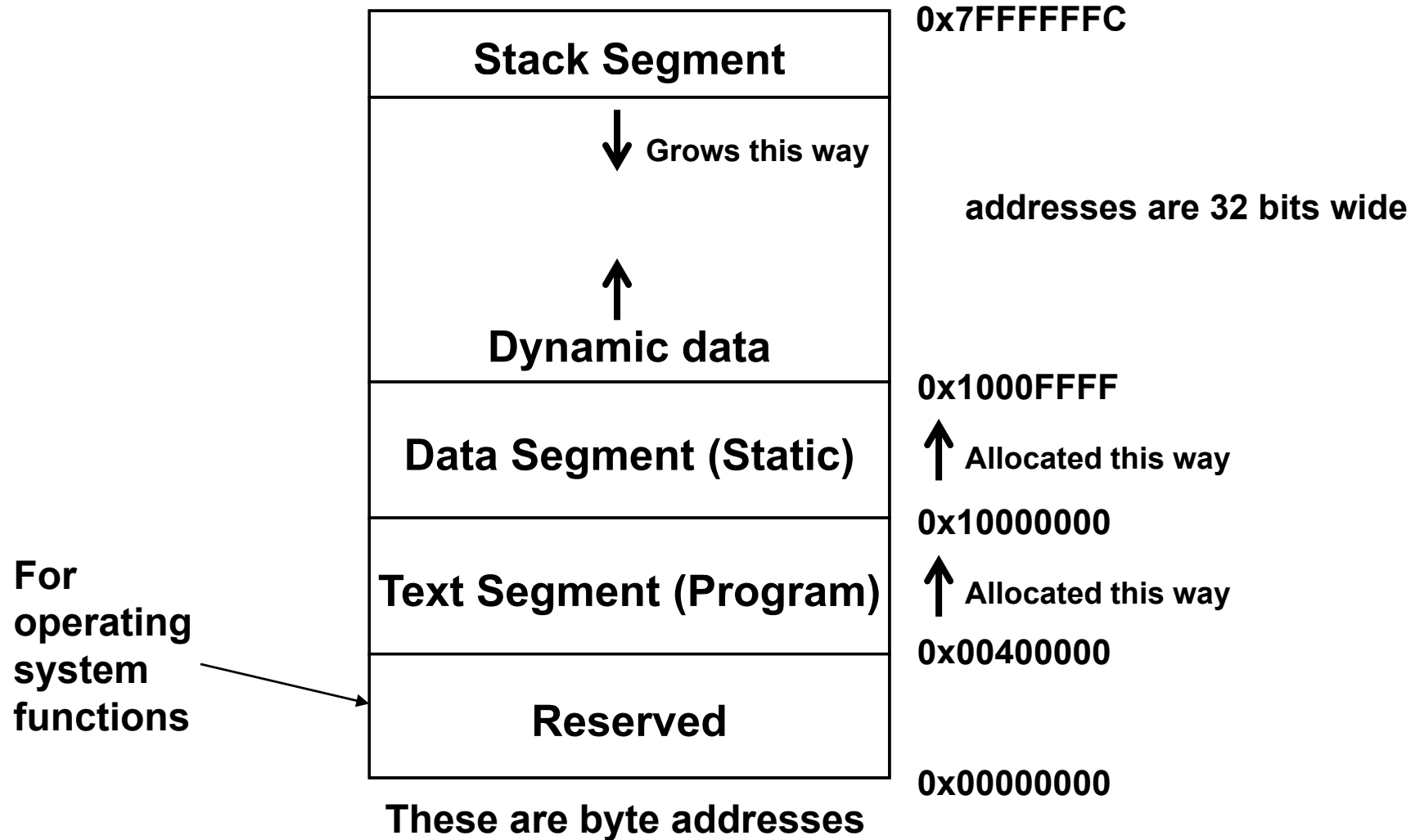


Figure B.5.1, Textbook, Appendix B, page B-21

# How to use MIPS registers(1/6)

---

- **Assignment statements & Arithmetic.**
- **MIPS allows three-operand instructions.**
- **Two are source & the third is a destination.**

**Example:**    `a = b+c;    →    add a, b, c`

`d = a-e;    →    sub d, a, e`

- **We have to get these variables in registers first.**

**Example: complex assignment**    `f = (g+h) – (i+j)`

**Here we need temporary locations to hold (g+h) and (i+j).**

**Using t0 & t1:**

```
add $t0, g, h    (see Textbook,  
add $t1, i, j    Sections 2.1-2.3, pp. 76-87)  
sub f, $t0, $t1
```

## How to use MIPS registers(2/6)

---

- Now suppose `f`, `g`, `h`, `i`, `j` are in registers `$s0`, `$s1`, `$s2`, `$s3` & `$s4`.

- We could write,

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

} this is correct MIPS code.

- How do we get variable-values in these registers?

- Using data transfer instructions.

- `lw` → load word from an address.

`sw` → store word to an address.

- Memory to register ('`lw`') and register to memory ('`sw`').

- Memory address is supplied by the variable name.

```
lw $s1, g    # loads g in s1.
sw $s0, f    # stores s0 to f.
```

} correct MIPS code?

## How to use MIPS registers(3/6)

---

- Addresses associated with 'lw' and 'sw' must be multiples of 4. That is how the memory is built.
- Alignment restriction: variables that are 1 word wide must start at a word boundary.
- Compiling an assignment statement when one operand is in memory.

Example:  $g = h + A[8];$

Let  $g$  and  $h$  be in  $\$s1$  and  $\$s2$  as before.

Let the start address (base) of  $A$  be in  $\$s3$ .

Address of  $A[8]$  is base address of  $A + 32$  (note:  $32 = 8 \cdot 4$ )

```
lw   $t0, 32($s3)
add  $s1, $s2, $t0 } 32 is called an 'offset'
```

Note how  $\$s3$  is used in the 'lw' instruction.

# How to use MIPS registers(4/6)

---

- When a register is specified within a pair of parenthesis, the contents of that register are treated as an address.
- Address of A[8] is contents of `$s3` plus 32 (an offset).

- Example:  $A[12] = h + A[8]$ .

Here `h` is in `$s2` and `$s3` has base of `A`.

```
lw  $t0, 32($s3)    ← t0 gets A[8]
```

```
add $t0, $s2, $t0   ← t0 = h + t0
```

```
sw  $t0, 48($s3)    ← A[12] = t0
```

- Load-Store architecture: only 'load' and 'store' refer to memory addresses.
- Instruction 'add' cannot refer to an address.

# How to use MIPS registers(5/6)

---

- **Most programs have more variables than registers in a machine.**
- **Compiler tries to keep the most frequently used variables in registers.**
- **Other variables are kept in memory**
- **Use load and store to work with them**
- **This process of keeping less commonly used variables in memory is called “spilling registers”**
  
- **By not allowing ‘add’ instruction to refer to a memory address, we can efficiently implement ‘add’, ‘load’, ‘store’ instructions.**
- **‘add \$t0, variable’ cannot be implemented efficiently. Not allowed in MIPS.**

# How to use MIPS registers(6/6)

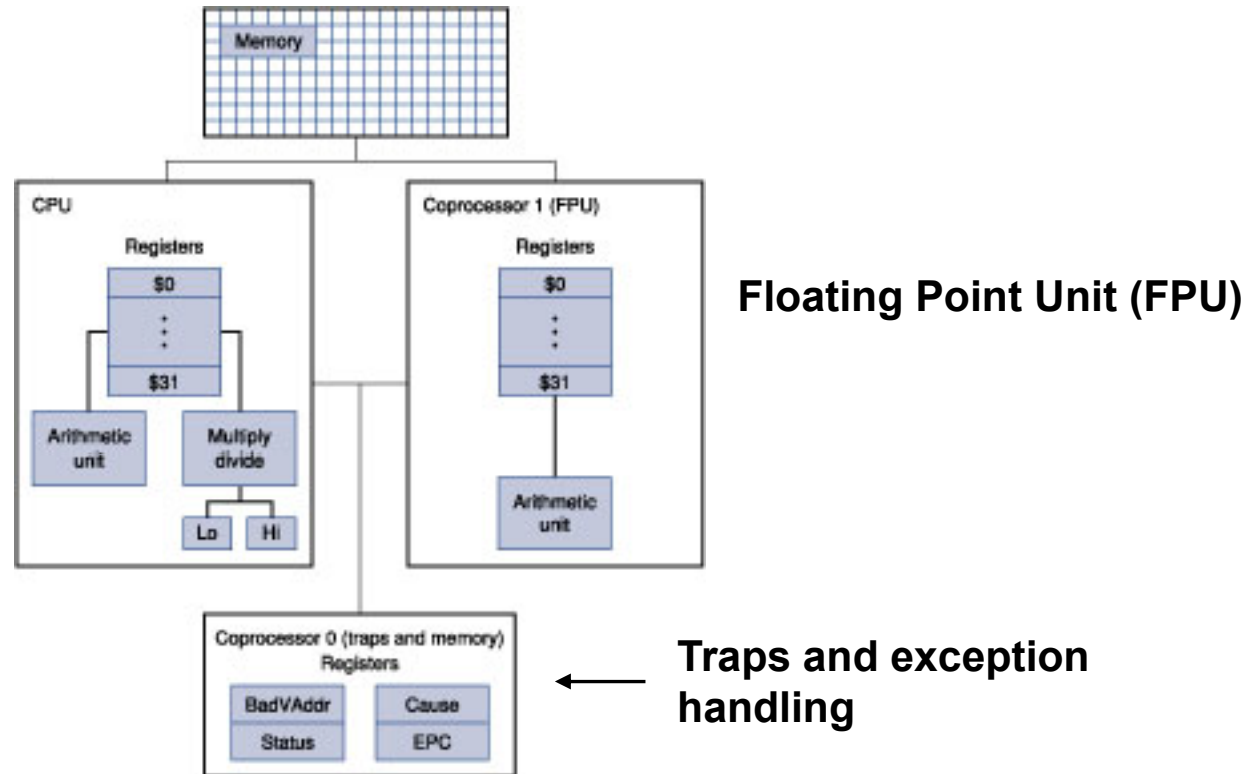


FIGURE B.10.1 MIPS R2000 CPU and FPU (Textbook, Appendix B, page B-46)

- These days FPU is part of the processor chip.
- Sequencing /control unit is not shown.
- This picture describes registers of MIPS.