Lecture 7

25-01-2021

Why Does RISC Lead to Improved Performance?

- Larger number of registers (visible and invisible)
  - **Also much larger cache**
  - **Decreases data traffic to memory.**
  - **Remember memory is the bottleneck.**
- Register-Register architecture leads to more uniform instructions:
  - **Efficient pipelining becomes possible.**
- However, increased instruction memory traffic:
  - Because of larger number of instructions .

Does Any Aspect of RISC Lead to Some Performance Degradation?

- Suppose we need to add 2 values and store the results back to memory.
- **In CISC:**

```
addm  1000, 1004, 1008
```

  - It would be done using a single instruction.
- **In RISC:**

```
load  R1, 1004
load  R2, 1008
add R3, R1, R2
store 1000,R3
```

  - 4 instructions would be necessary.
- Increased instruction memory traffic

CISC versus RISC: Summary

CISC	RISC

CISC vs. RISC

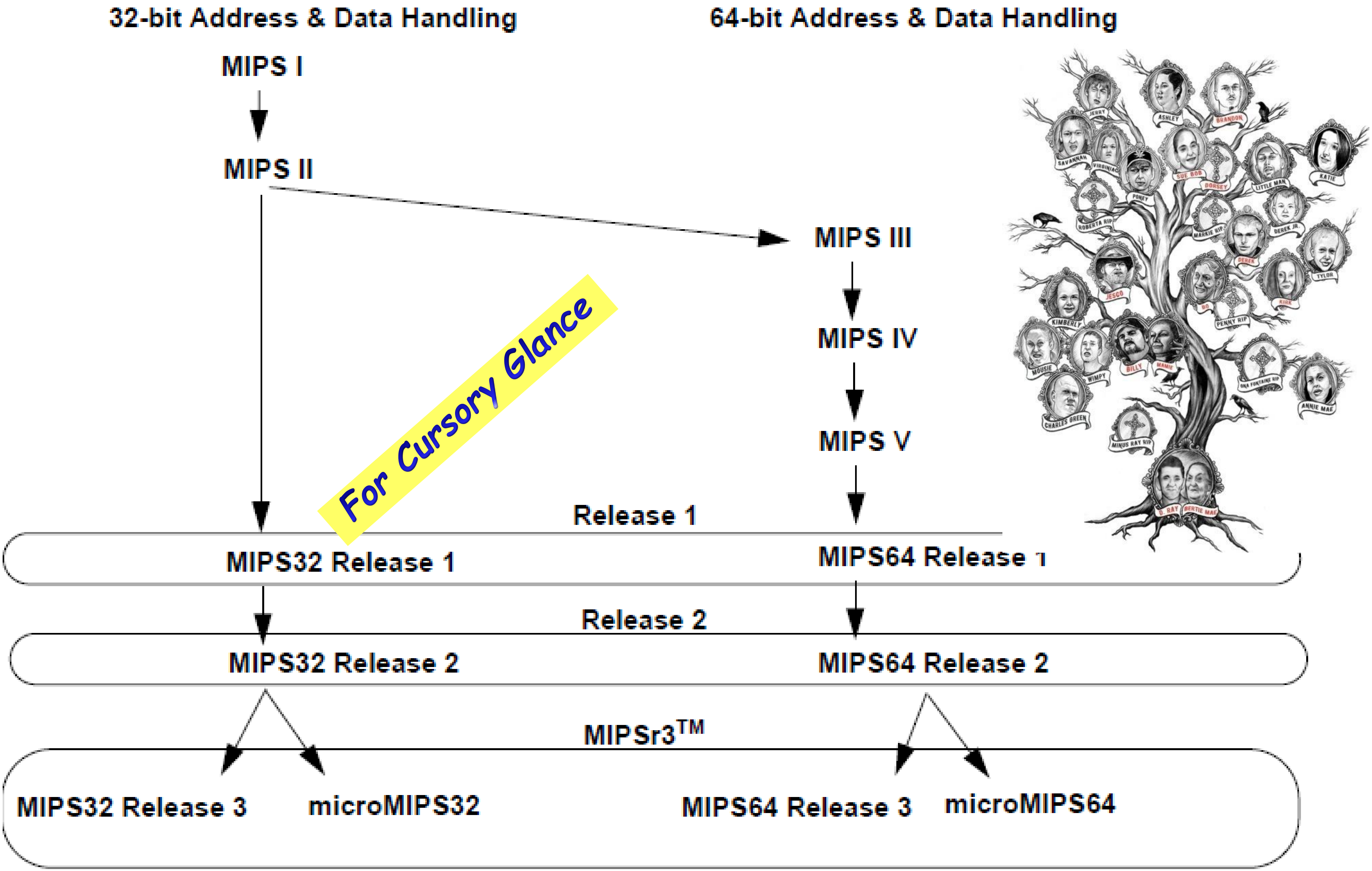
	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC1	Stanford MIPS
Year introduced	1973	1978	1978	1980	1981	1983
# instructions	208	303	270	120	39	55
Microcode Mem	54KB	61KB	17KB	0	0	0
Instruction size	2 to 6 B	2 to 57 B	1 to 3 B	4B	4B	4B
Addressing Modes	Reg-reg Reg-mem Mem-mem	Reg-reg Reg-mem Mem-mem	Stack	Reg-reg	Reg-reg	Reg-reg

MIPS Architecture

MIPS

- **M**icroprocessor **Without** **Interlocked** **P**ipeline **S**tages (MIPS):
  - One of the first implementation of RISC architectures
  - Grew out of research at Stanford University
- MIPS computer system founded in 1984:
  - R2000 introduced in 1986.
  - **Licensed designs rather than directly manufacturing processor or selling the design.**

MIPS family tree



32-bit Address & Data Handling

64-bit Address & Data Handling

MIPS I

MIPS II

MIPS III

MIPS IV

MIPS V

Release 1

MIPS32 Release 1

MIPS64 Release 1

Release 2

MIPS32 Release 2

MIPS64 Release 2

MIPSr3™

MIPS32 Release 3

microMIPS32

MIPS64 Release 3

microMIPS64

For Cursory Glance

1981 Dr. John Hennessy at Stanford University leads research on RISC principles.

1984 MIPS Computer Systems, Inc. co-founded by Dr. John Hennessy, Skip Stritter, and Dr. John Moussouris

1986 First product ships: R2000 microprocessor, Unix workstation, and optimizing compilers

1988 R3000 microprocessor

1989 First IPO in November as MIPS Computer Systems with Bob Miller as CEO

1991 R4000 microprocessor

1992 SGI acquires MIPS Computer Systems. renames it to MIPS Technologies, Inc. (a wholly owned subsidiary of SGI)

1994 R8000 microprocessor

1994 Sony PlayStation released, using an R3000 CPU with custom GTE coprocessor

1996 R10000 microprocessor; Nintendo 64 released, incorporating a cut down R4300 processor.

1999 Sony PlayStation 2 released, using an R5900 cpu with custom vector coprocessors

2002 Acquires Algorithmics Ltd, a UK-based MIPS development hardware/software and consultancy company.

2005 Acquires First Silicon Solutions (FS2), a Lake Oswego, Oregon

2007 MIPS Technologies acquires Portugal-based mixed-signal intellectual property company Chipidea February, 2009: MIPS Joins Linux Foundation

2009 Chipidea is sold to Synopsys.

2009 Android is ported to MIPS

2009 MIPS Technologies joins the Open Handset Alliance

2010 Sandeep Vij appointed as CEO

2011 MIPS introduces the first Android-MIPS based Set top box at CES

2012 MIPS Technologies ports Google's Android 4.1, "Jelly Bean". With Indian company Karbonn Mobiles announces world's second tablet running Android 4.1.

2013 MIPS Technologies is sold to Imagination Technologies.

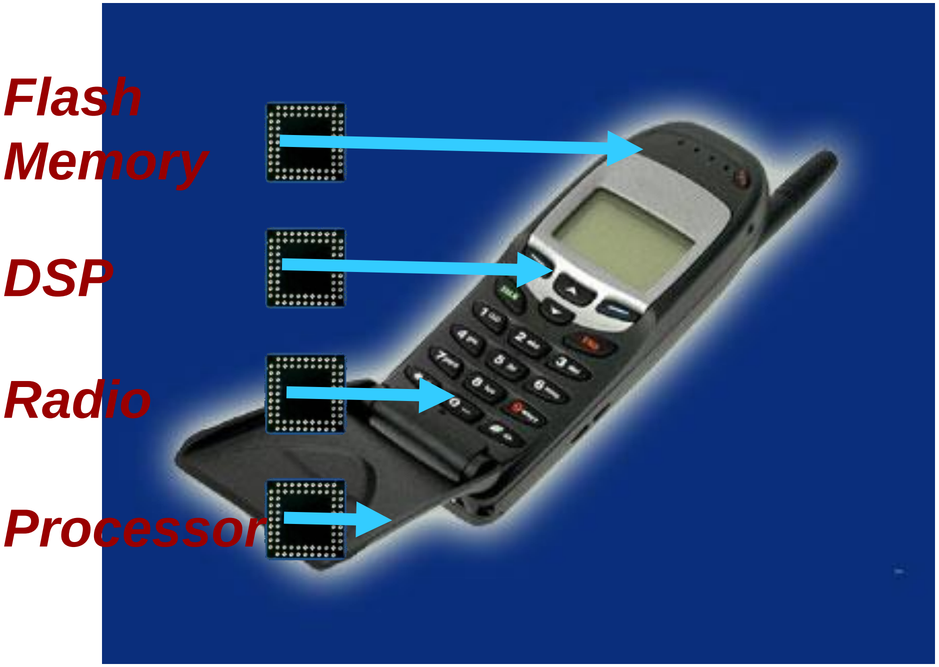
For Cursory Glance

Commercial Success of MIPS

- **Popularly used as IP-cores (building-blocks) for embedded processor (SoC) designs.**
  - Both 32-bit and 64-bit basic cores were offered --- the design is licensed as MIPS32 and MIPS64.
  - MIPS cores have been commercially successful --- used in many consumer and industrial applications.
- MIPS cores could be found in:
  - Cisco and Linksys routers, cable modems and ADSL modems, smartcards, laser printer engines, set-top boxes, robots, handheld computers, Sony PlayStation 2 and Sony PlayStation Portable, Nintendo….
- **In cell phone/PDA applications, the MIPS core was unable to displace the incumbent, competing ARM core**

Digression: What is SoC?

Yesterday



Flash Memory

DSP

Radio

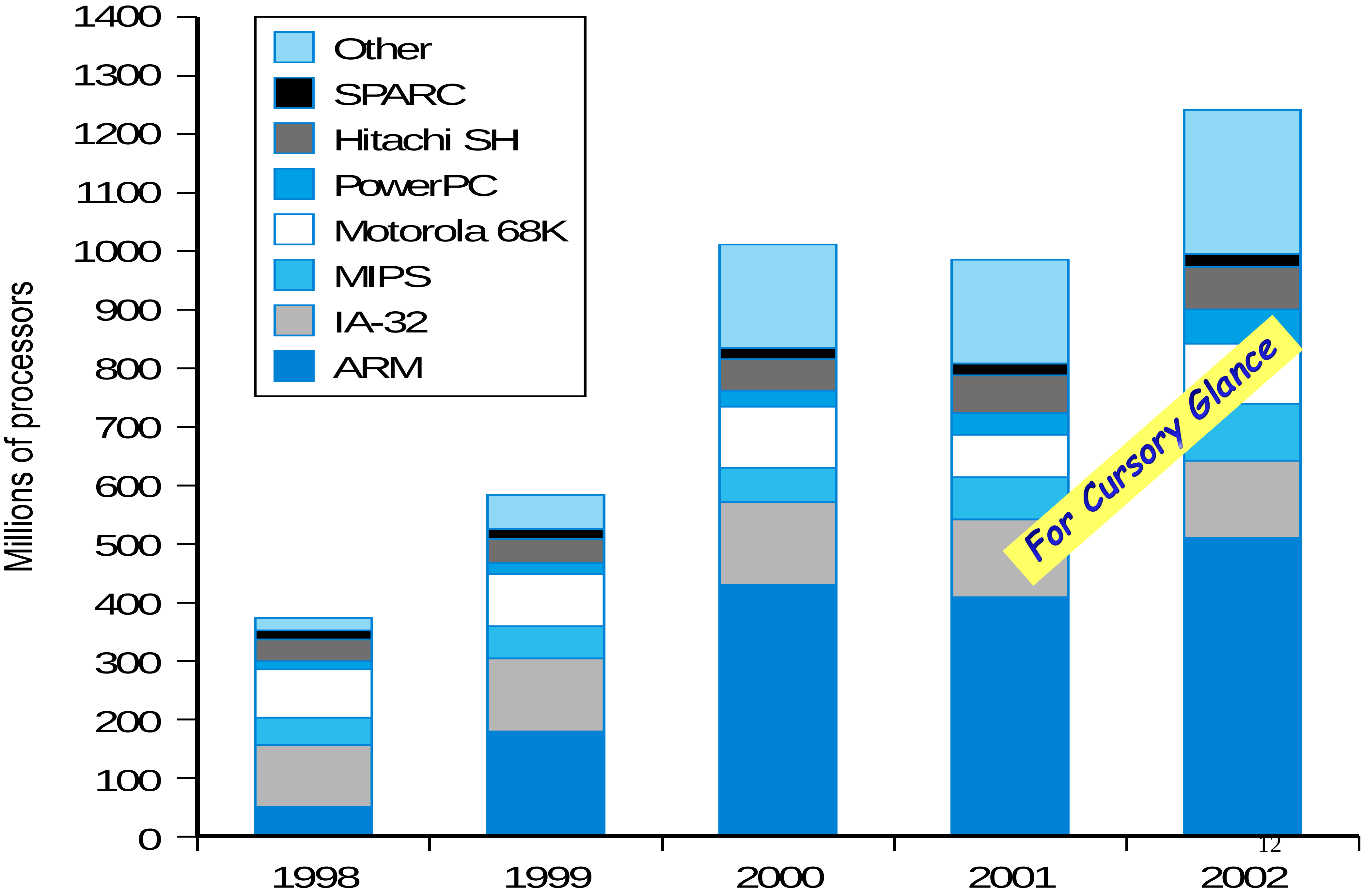
Processor

Today

Single Chip

- 5~8 Processors
- Memory
- Graphics
- Bluetooth
- GPS
- Radio
- WLAN

MIPS Processor was popular…



For Cursory Glance

Gadgets shipped with MIPS cores

- Cable Modems 94%
- DSL Modems 40%
- VDSL Modems 93%
- IDTV 40%
- Cable STBs 76%
- DVD Recorder 75%
- Game Consoles 76%
- Office Automation 48%
- Color Laser Printers 62%
- Commercial Color Copiers 73%

For Cursory Glance

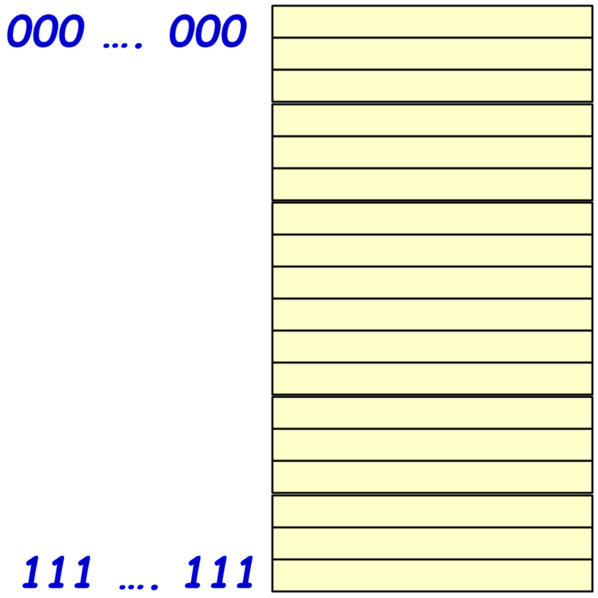
Source: Website of MIPS Technologies, Inc., 2004.

MIPS: A RISC example

- Small set of simple instructions:
  - 111 instructions
- 32 32bit registers
- 2   128KB high speed cache
- Constant 32bit instruction length
- One cycle execution time per stage (CPI=1)

MIPS

- MIPS memory:
  - Byte addressable with 32-bit address.
- MIPS provides 4 broad classes of instructions:
  - ALU operations
  - Load/Store
  - Branches and jumps
  - Floating point operations



MIPS Instruction Set

- 25 branch/jump instructions
- 21 arithmetic instructions
- 15 load instructions
- 12 comparison instructions
- 10 store instructions
- 8 logic instructions
- 8 bit manipulation instructions
- 8 move instructions
- 4 miscellaneous instructions

For Cursory Glance

MIPS Design Principles

- **Simplicity calls for regularity:**
  - All instructions 32 bits
  - All ALU instructions have 3 operands
  - Limited number of ALU instruction formats: R, I, J
    - **Register-register,**
    - **register-immediate,**
    - **Jump**

Variables allocated to Registers

- Registers are numbered from 0 to 31
- Each register can be referred to by its number or name
- Register Number referencing:

$0, $1, $2, … $30, $31

Registers Referred by Name

- Example:

  $16 - $23 ➔ $s0 - $s7

  (correspond to variables)

  $8 - $15 ➔ $t0 - $t7

  (correspond to temporary variables)

  Later will explain other 16 register names

- Referring by names usually make code more readable

MIPS Registers

Name	Register Number	Usage	Preserved by callee
$zero	0	hardwired 0	N/A
$v0-$v1	2-3	return value	no
$a0-$a3	4-7	arguments	no
$t0-$t7	8-15	temporary values	no
$s0-$s7	16-23	saved values	YES
$t8-$t9	24-25	more temporary values	no
$gp	28	global pointer	YES
$sp	29	stack pointer	YES
$fp	30	frame pointer	YES
$ra	31	return address	YES

MIPS Instruction Set

- Only **Load** instruction can read an operand from memory
- Only **Store** instruction can write an operand to memory
- Typcial RISC calculations:
  - Load(s) to get operands from memory into registers
  - Calculations performed only on values in registers
  - Store(s) to write result from register back to memory

MIPS Assembly Language Examples

- **ADD $1,$2,$3**
  - Register 1 = Register 2 + Register 3
- **SUB $1,$2,$3**
  - Register 1 = Register 2 - Register 3
- **AND $1,$2,$3**
  - Register 1 = Register 2 AND Register 3
- **ADDI $1,$2,10**
  - Register 1 = Register 2 + 10
- **SLL $1, $2, 10**
  - Register 1 = Register 2 shifted left 10 bits

MIPS Assembly Language Examples

- **LW $1,100**
  - Register 1 = Contents of memory location 100
  - Used to load registers
- **SW $1,100**
  - Contents of memory location 100 = Register 1
  - Used to save registers
- **LUI $1,100**
  - Register 1 upper 16-bits = 100
  - Lower 16-bits are set to all 0's
  - Used to load a constant in the upper 16-bits
  - Other constants in instructions are only 16-bits, so this instruction is used when a constant larger than 16-bits is required for the operation

MIPS Assembly Language Examples

- **J 100**
  - Jumps to PC+100
- **JAL 100**
  - Save PC in $31 and Jumps to PC+100
  - Used for subroutine calls
- **JR $31**
  - Jumps to address in register 31
  - Used for subroutine returns

MIPS Assembly Language Examples

- **BEQ $1, $2, 100**
  - If register 1 equal to register 2 jump to PC+100
  - Used for Assembly Language If statements
- **BNE $1, $2, 100**
  - If register 1 **not equal** to register 2 jump to PC+100
  - Used for Assembly Language If statements

MIPS Labels

- Instead of absolute memory addresses:
  - Symbolic labels are used to indicate memory addresses in assembly language
- Assembly Language Programs are easier to modify and understand when labels are used…
- **Example:**

```
Loop: lw   R1, 1004
      lw   R2, 1008
      add R3, R1, R2
      sw 1000,R3
      J Loop
```

  - Location LOOP is address 250 --- Use label LOOP instead of jump address 250

First MIPS Program:
A = B + C;

```
LW  $2, B     ;Register 2 = value at B

LW  $3, C     ;Register 3 = value at C

ADD $4, $2, $3 ;Register 4 = B+C

SW  $4, A     ; A = B + C
```

The Constant Zero

- $R0 (or $zero) is always contains constant 0
  - Cannot store any other value. What is the use?
- Useful for synthesizing common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

  - **There is no separate Mov instruction**
  - **add R3,R0,3** #move 3 to R3
- **R0 helps in reducing the number of instructions.**

MIPS: Addressing modes

- Only immediate and displacement modes supported, besides register mode.
- Register indirect achieved by placing 0 in displacement field.
  - LW R4, 0(R1) //[R4]<--[[R1]]
- Absolute addressing achieved by using R0 as the base register.
  - LW R1, 100(R0) //[R1]<- [100]

MIPS Arithmetic

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)
- Example

C code: **A = B + C**

Assume A,B,C in $s0, $s1, and $s2

MIPS code: **add $s0, $s1, $s2**

Temporary Variables

- Intermediate results of expressions are stored in temporary variables.

- C code: **f = (g + h) - (i + j);**

Write MIPS code: *Assume* f, g, h, i, j are in $s0 through $s4 respectively

Data Transfer

```
lw $t0, 8($s3) # base address in $s3, offset 8
Addi   $t0,$t0,100
sw $t0, 48($s3) # store word
```

…	…
12	0101
8	0110
4	0010
0	1001
Address	Memory

Byte - 8 bits

Word - 32 bits

Memory access in words

Address to byte level

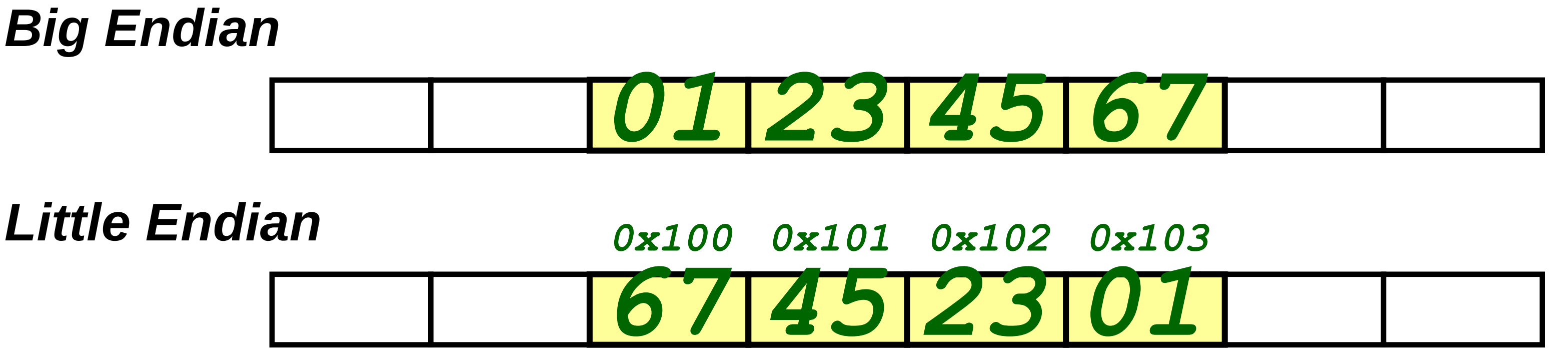
2^{32} bytes with byte addresses from 0 to 2^{32}

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

what are the least 2 significant bits of a word address?

Byte Ordering

- **Big Endian**
  - Least significant byte has highest address
- **Little Endian**
  - Least significant byte has lowest address
- **Example:**
  - Variable **v** has 4-byte representation **0x01234567**
  - Address given by **&v** is **0x100**



Big Endian

01 23 45 67

Little Endian

0x100 0x101 0x102 0x103

67 45 23 01

Exercise

- Consider **0xFF00AA11**

1000	11
1001	AA
1002	00
1003	FF

Little Endian

1000	FF
1001	00
1002	AA
1003	11

Big Endian

1. The address for a group of bytes is the smallest address of the four.
2. What goes in that byte is:
    1. Big Endian: the big end
    2. Little Endian: the little end
3. The remaining three bytes are filled in sequence.

Memory Operand Example 1

- C code:

  ```
  g = h + A[8];
  ```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Load and Store

- All arithmetic instructions operate on registers
- Memory is accessed through load and store instructions
- An example C code: **A[12] = h + A[8];**
  Assume that $s3 contains the base address of A
- $s2 contains h

  Write MIPS code:

- Note: arithmetic operands are registers, not memory!

invalid: ~~**add 48($s3), $s2, 32($s3)**~~

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction